VOLUME 05 | ISSUE 05

digit FastTrack
YOUR HANDY GUIDE TO EVERYDAY TECHNOLOGY

To Creating iPhone apps

We show you how to make iPhone apps that you might be able to sell to some of the 50+ million iPhone users out there

CHAPTERS

Fast Track to Creating iPhone Apps

A 9 9 Media Publication

digit
YOUR TECHNOLOGY NAVIGATOR
thinkdigit.com

Fast Track

to

Creating iPhone apps

Credits

The People Behind This Book

Editorial

Editor	Robert Sovereign-Smith
Head-Copy Desk	Nash David
Writers	Nishith Rastogi, Rahil Banthia, Nilesh Agarwal

Design and Layout

Lead Designer	Vijay Padaya
Senior Designer	Baiju NV
Cover Design	Santosh Kushwaha

May 2010

Free with Digit. Not to be sold separately. If you have paid separately for this book, please email the editor at editor@thinkdigit.com along with details of location of purchase, for appropriate action.

Contents

Contents

Introduction

The world as we know it changed on January 9, 2007 with the launch of the iPhone. It introduced and made touch interfaces the norm among smartphones, and smartphones a norm among mobile phones.

The already excellent functionality of the premium mobile device was further extended when Apple released the iPhone SDK on March 06, 2008 allowing developers to put their favourite and most profitable functionality on the iPhone. The iPhone apps made by any developer can be sold on the online app store by Apple where 70 per cent of the revenue from app sales goes to the developer and the remainder goes to Apple. This means big business, App Store is estimated to bring more than $1 billion to Apple alone annually or approximately a lucrative market worth about Rs. 10,000 Crore for the developers. There are countries whose annual GDP is less than that.

The iPhone and its associated SDK made things a lot easy for mobile developers who were earlier plagued with the problem of fragmentation and a lack of single global distribution channel. With the iPhone SDK, a developer for the mobile platform can write an app that he is assured of running smoothly and in a desired fashion across millions of devices globally in circulation. In addition, revenue generation is far easier and hence motivated the development of polished professional applications for a mobile phone.

Today with over 50,000 apps in the App Store you can do anything on your iPhone from playing tetris to preliminary medical diagnoses and yes, this time you can even learn zen. The iPhone SDK has become an indespensible tool in the arsenal of any developer who is looking towards the mobile platform, which today every developer should. In india there are 10 mobile phones for every single desktop and laptop combined and there are more than quadruple the number of internet enabled mobile phones than a computer.

Writing applications for mobile phones is challenging given the limited processing power, memory, screen estate. iPhone makes it further

complicated by forcing only one application to run except the OS though makes user interaction far simpler with its awesome multi-touch capable capacitive screen.

We walk you through the very beginning from obtaining the SDK to finally making a usable application utilising multi-touch, audio, video and more. Hold tight, we are beginning to start a journey on fun-filled code adventure.

1 Getting Started

1.1 Obtaining the iPhone SDK

As hard as it might be to believe, you can only develop for the iPhone on a Macintosh (the closest is a Mac OS X virtual machine). Currently, the iPhone SDK comes bundled along with its IDE (Integrated Development Environment) which includes its interface builder all rolled into one nice package Called the Xcode. It is available as a free download from Apple's web site and comes in two flavours:

- The iPhone SDK with Xcode 3.1.4 for Leopard (10.5)
- The iPhone SDK with XCode 3.2.1 for Snow Leopard (10.6)

It's important to note that though any version of Leopard will allow you to install the XCode only 10.5.7 upwards will let your install the iPhone SDK with it also.

Download your SDK for free

Once finished with the registration, you're allowed to download any or both of the flavours mentioned above.

Installing the SDK is very simple, and like any other Mac OS package you only have to double-click on your freshly downloaded DMG file. This launches the installer, and after a couple of dialog boxes later your Mac Machine is ready to roll out iPhone code.

Your Xcode application is located under the /Developer/Application menu which is intuitive. The iPhone SDK also comes along with an iPhone simulator that runs your applications in a simulated environment, so you don't have to test your app each time on a physical iPhone/itouch device. For this, you need an Apple's Developer Licence. You also get access to the dashcode, which is the Integrated Development Environment that allows you to code for web-based applications for the iPhone platform.

The Interface Builder is similar to the Visual Basic form design tool. With the Interface Builder, you can develop polished-looking GUIs with ease have radio buttons, text fields and so forth. Finally, you have instruments available with the XCode that lets you analyse how well your code is running in real-time with respect to CPU usage and memory leaks. Before we move any further, it is important to understand the limitations as well as features of the iPhone simulator. Resembling the real physical device, the simulator lets you do the following and test your application for the following:

Screen rotation – left, top, and right
Gesture Support for:
Tap
Touch and Hold
Double Tap
Swipe
Flick
Drag
Pinch
Low – memory warning simulations

Limitations

Obtaining real time location data – it returns only a fixed present coordinate

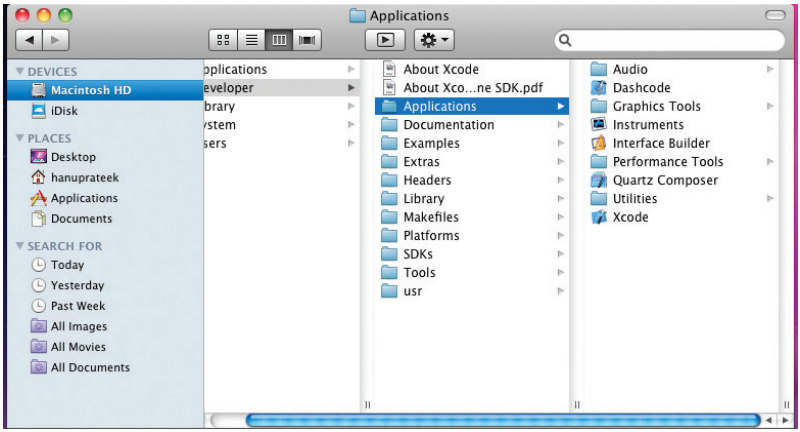
Given the fact it is a simulator, it can't make phone calls or receive them, the same with messages.

You can't access the accelerometer.

1.2 Launch the XCode

All the iPhone SDK packages or tools are installed in the /Developer/Application section, from there select XCode by double-clicking on its icon. You can, of course, also fire the ever helpful spotlight and search for Xcode.

At the launch of Xcode you will be greeted with a welcome screen. Next,



Locate your Xcode

click on the "New Project" which is accessible via the file menu which will bring up the "New Project Assistant".

Welcome Screen

Your first job over here is to select between development for iPhone or the Mac OS X. For the Mac OS X there will be tons of categories available like

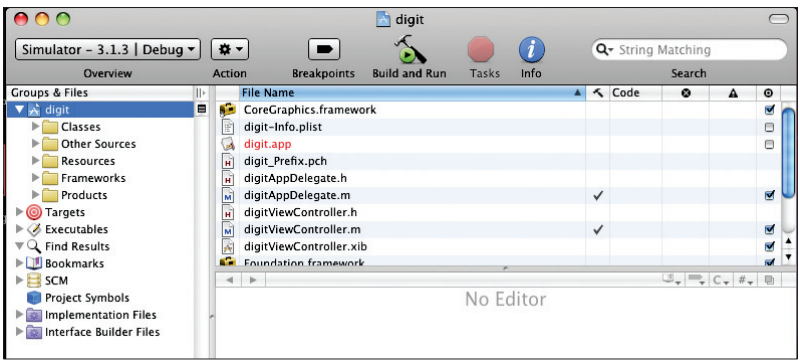
Application, Framework, Java etc but fortunately for you there will be only one of the iPhone OS, called "Application".

Within the "Application" category you will be presented with the following choices:

Navigation-based Application
OpenGL ES Applications
Tab Bar Applications
Utility Applications
View-Based Application
Window-based Application

As usual we will start with a Hello World application for which you should select the "View-Based Application". This particular template is iPhone equivalent of the bare-bone framework, but is capable enough to take you to a fine start. Upon selection of the "View-Based Application" you will be asked to enter a name and storage location for the project. We entered "Digit" as the name.

In your editor window "Groups and Files" pane is the one that needs the most explanation and your attention. To further simplify matters as of now,



Your window into the world of Apple Development

just consider the headings under the "Digit" , assuming your application is called "Digit". This will contain five further subfolders namely

Classes: Majority of your code belongs here, you can further create sub-folders under this to keep some hierchy in your code. This will contain all your Objective-C code.

Other Sources: This is the place where your non-objective code resides. It

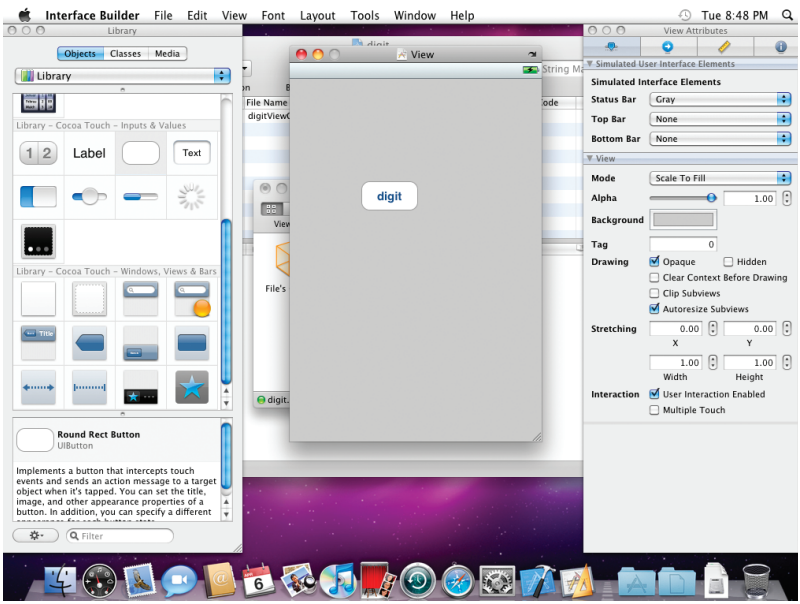
has two files one is called the “Digit_Prefix.pch”. Pch stands for precompiled header, which is basically the external framework precompiled to save you compilation time. The other being main.m which calls your application's main() method.

Resources: This is the non-code part of your project. Yes, you guessed it right, this contains the images, sound files, movie files, text files or whatever you want to include in your application.

Frameworks: This contains the libraries needed to run your app, most of the libraries are linked by default so you won’t need much meddling around with it.

Products: This is the end result of your application. IF you open it you would see a file called “Digit.app” which is the resultant app of your project. Initially it would be in red, indicating it doesn’t exist, once you compile and build your project successfully once the red colour will go.

Enough of the ugly source files, let's do something that the iPhone is known for. We're talking about beautiful seamless GUIs. For this you would need to fire up the interface builder. You can open it from within the XCode window by



Apple’s Counter part of the Visual Basic form builder

going to your "Resources" group and then opening "DigitViewController.xib". Doing this will result in the firing of Interface Builder

A point to note is that though the file extension is called xib, it is a very recent change, it was originally called "nib" and even now most of the Apple's Documentation refer it that way only. Hence even in this guide we will be referring to them as NIB files.

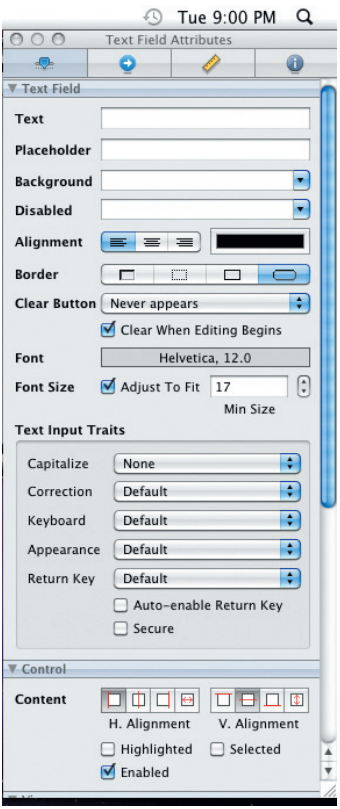
Here you can experiment and add delete visual elements. The interface builder lets you do what you can technically do via code also but in a far more intuitive way. For example instead of writing the following

```
UIButton *digitButton = [[UIButton alloc] initWithFrame:aRect];
```

you can simply drag and drop a button, and also easily modify its shape, size, color and other properties from the interface builder. To change the attribute of any object, select the object and then press `[Command] + [1]` or go to `Tools > Inspecter` to launch the attribute window. The main graphical layout of your application is shown in a window called the "View Window."

The UIbutton is part of the "Cocoa Touch" objects, the GUI layer of your code. Each object that you drag to your frame creates an instance of that class in your application. In case you want to label the button, you can either modify its text property or you can drag and drop a "Label" Object into the frame (the View Window). To modify its text just double-click and edit. You can always drag and drop it around the view window to position it where you want.

If at this point you are getting really impatient, you can select Save from the File menu of the interface builder. In Xcode, go to `Build > Build and Run`, and then if there are no errors in your code (there should be none till this point), the



Tweak your GUI to your heart's content.

Xcode will compile, build and launch the iPhone simulator and run your application. Which in this case should display the label (and the button if you added).

Congratulations on your first iPhone app! You can now click on the home button in the simulator and you will see the home screen which will be pretty blank right now, but for a couple of icons and your brand new app with the name "Digit" below it.

Pressing the home button in the simulator when your app is running in it will also install in the virtual iPhone on your Mac.

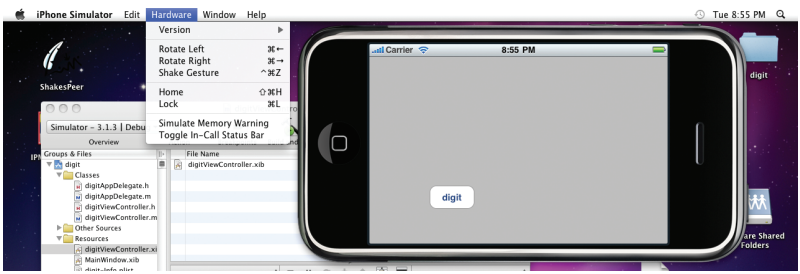
If you press the [Command]+ right/left key combination, you can change the orientation of your iPhone simulator. At this point you will notice that your display hasn't reacted to the change in orientation and your app just looks like it has been rotated by 90 degrees. To make it respond go to XCode and open the `DigitViewController.m` file.

Search for `shouldAutorotateToInterfaceOrientation:` and modify the code block associated with it to the following. You might have to remove the comment markers, they are the same as in C/C++, namely `//`

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterf
aceOrientation)
   interfaceOrientation {
          return YES;   // Returning Yes will enable the
orientation change
   }
```

Now if you build and run your application you will notice on changing the orientation in the simulator, your display responds and your "Label" and "Text" still are displayed parallel to your eyes, irrespective of the orientation.

As displayed above, you will notice that though the label rotated as you changed the orientation, the alignment doesn't look pretty anymore. For this,



Instead of Using keyboard shortcut, you can use hardware menu

you will have to modify the attribute of the label, that can be done as explained above, by opening the `Tools > Size Inspector`. Here, alter the "Autosizing" attribute. You can also anchor your visual object to any of the screen edges.

1.3 Creating actions

Now it's time to get back to writing some code. Let us try and do something on our screen when you click on a button in your UI. For this we will first create an action and then link that action to the click of the window.

So gear up and focus back on the tiny window called the "digitViewControlloer.xib" this will be one of the three windows in your interface builder. This contains only three things inside it namely

- File's Owner
- First Responder
- View

Now select File's Owner and then open the Identity Inspector form the tools menu. Add a Class Action and name it `bnClicked:` , this has created a class action called `bnClicked:`. Don't miss out on the colon at the end.

Now drag the button from the "View" window to the "File's Owner" in the `digitViewController.xib`; this should result in a pop-up containing the `bnClicked:` action that you just created. This links the click of the button with that action.

Now to actually do some action, you will have to write its required code, this can be done by editing two files. The first being `DigitViewController.h` where you would add a header declaration for the bnClicked:

It should resemble something like this:

```
//  digitViewController.h
//  Created by Subiet
//  Created cause he was bored

#import  <  UIKit/UIKit.h  >
@interface digitViewController : UIViewController {
}
-(IBAction) bnClicked:(id)sender;
@end
```

The line in bold, is your declaration. Now move towards the `DigitViewController.m` file, this is where you would write the actual action code:

```
- (void)dealloc {
```

```
    [super dealloc];
}

-(IBAction) bnClicked:(id)sender {

        UIAlertView *alert =    [[UIAlertView alloc]
initWithTitle:@"Hello Everyone"
message: @"Reading Fast Track is fun"
delegate:self
cancelButtonTitle:@"Awesome"
otherButtonTitles:nil, nil];
    [alert show];
    [alert release];
}

@end
```

Even if you don't understand the code above, don't be discouraged. You'll get a hang of it as you proceed. To build-run your application, when you click on the button in your simulator, you will notice a nice little pop-up saying `Hello Everyone; Reading Fast Track is fun` which can be closed by clicking the "Awesome" button.

The iPhone has a 480x320 resolution at 163 PPI, but while designing your application you should design it ideally to run in a 460x320 environment to leave enough space for the status bar. Though it can be hidden and you can use the whole 480x320 pixels, it is not recommended under general circumstances.

2 Objective C Basics

Objective-C is basically just an extension to the standard ANSI C language, except that it's an object-oriented programming language such as Java or C#. It's used primarily by Apple for programming Mac OS X and iPhone applications. If you're familiar with C, you'll find this easy to learn. At the same time, novices won't find this a horrendous task. The source code for any iPhone application is contained in two types of files – header files (.h) and implementation files (.m).

2.1 Preprocessor directives

Just as all your C and C++ include preprocessor directives use the `#include` statement, Objective-C files use `#import`. The top of your BasicClass.h file will contain the #import statement as follows:

```
#import < Foundation/Foundation.h >
@interface BasicClass : NSObject {
}
@end
```

The Objective-C compiler ensures that any header file is included at most only once. The syntax to import a header file from one of the framework APIs is to specify the header filename using angle brackets < > in the #import statement. If the header file is already within your project, modify the syntax to use double quotes ("") instead of the angle brackets. For example, in the `BasicClass.m` file, we will import the `BasicClass.h` file as follows:

```
#import "BasicClass.h"
@implementation BasicClass
@end
```

Methods

All the functions that you define in a class are referred to as methods.

Classes

Most of time in object-oriented programming goes in dealing with different classes. In Objective-C, classes are declared using the @interface compiler directive:

```
@interface BasicClass : NSObject {
}
```

You need to declare all the classes in the header file (.h) without any implementation defined. This code essentially declares a class named BasicClass , and this class inherits from the base class named NSObject, which is the root class of most Objective-C classes. It defines the basic interface of a class in Objective-C and contains methods common to all classes that inherit from it. It also provides the standard memory management and initialization framework used by most objects in Objective-C as well as reflection and type operations.

If otherwise that class we wanted to declare was a View Controller class, we will inherit from the UIViewController class:

```
@interface BasicClassViewController : UIViewController
{
}
```

Initializers

While creating an instance of a class, you can also initialise it at the same time. For example, in the running example that we've been using, you had this statement:

```
BasicClass *sc = [[BasicClass alloc] init];
```

We use the alloc keyword here to allocate memory for the object, and when an object is returned, we call the `init` method on the object to initialise the object. We've however not defined a method named init in our BasicClass. But this init method automatically comes from the NSObject class, which we've told you is the base class of most classes in Objective - C. This method is known as an initialiser. You can also create additional initialisers by define other methods that begin with the init word.

```
#import <Foundation/Foundation.h>
@class SecondClass;
@interface BasicClass : NSObject {
SecondClass *secondClass;
float rate;
NSString *name;
}
-(void) doSomething;
-(void) doSomething:(NSString *) str;
-(void) doSomething:(NSString *) str
withAnotherPara:(float) value;
+(void) alsoDoSomething;
```

```
- (id)initWithName:(NSString *) n;
- (id)initWithName:(NSString *) n andRate:(fl oat) r;
@property float rate;
@property (retain, nonatomic) NSString *name;
@end
```

There are two additional initializers: `initWithName:` and `initWithName:andRate:` in the code that we wrote. To provide the implementations for the two initializers you can use the following code:

```
#import "BasicClass.h"
@implementation BasicClass
@synthesize rate, name;
- (id)initWithName:(NSString *) n
{
return [self initWithName:n andRate:0.0f];
}
- (id)initWithName:(NSString *) n andRate:(fl oat) r
{
if (self = [super init]) {
self.name = n;
self.rate = r;
}
return self;
}
-(void) doSomething {
}
-(void) doSomething:(NSString *) str {
}
-(void) doSomething:(NSString *) str
withAnotherPara:(float) value {
}
+(void) alsoDoSomething {
}
@end
```

It is necessary to first call the init initialiser of the super (base) class because base class needs to be properly initialised, before you can initialise the current class in any initialiser implementation.

```
- (id)initWithName:(NSString *) n andRate:(float) r
{
```

```
if (self = [super init]) {
//
//
}
return self;
}
```

If a class is initialised properly, it returns a reference to self (hence the id type). However, if it fails, it returns nil. We've done that using this code in the `initWithName: initialiser implementation:`

```
initialiser:
- (id)initWithName:(NSString *) n
{
return [self initWithName:n andRate:0.0f];
}
```

If you have multiple initialisers, and each of them has different parameters, you should chain them by ensuring that they all call a single initialiser that performs the call to the super class' initialiser. This initialiser is called the designated initialiser.

Protocols

A protocol declares a programmed interface or a set of methods that any adopting class can choose to implement. The defining class of the protocol is expected to call the methods in the protocol that are implemented by the adopting class. Let's examine the UIAlertView class to understand protocols in detail:

```
UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:@"Hello"
message:@"This is an alert view"
delegate:self
cancelButtonTitle:@"OK"
otherButtonTitles:nil];
[alert show];
```

We are using the UIAlertView class by creating an instance of it and then calling its show method which displays an alert view with one button - OK. If you tap on the OK button, the alert view gets automatically dismissed. Additional buttons can also be displayed by setting the otherButtonTitles: parameter like:

```
UIAlertView *alert = [[UIAlertView alloc]
```

```
initWithTitle:@"Hello"
message:@"This is an alert view"
delegate:self
cancelButtonTitle:@"OK"
otherButtonTitles:@"No", @"Cancel", nil];
```

To determine which button was tapped by the user, you need to handle the relevant method(s) that will be fired by the alert view when the buttons are clicked. The UIAlertViewDelegate protocol handles these methods. They are:

- alertView:clickedButtonAtIndex:
- willPresentAlertView:
- didPresentAlertView:
- alertView:willDismissWithButtonIndex:
- alertView:didDismissWithButtonIndex:
- alertViewCancel:

To implement any of these methods in the UIAlertViewDelegate protocol, you need to ensure that your class, in this case the View Controller, conforms to this protocol. This is made possible by using angle brackets (<>) as follows:

```
@interface UsingViewsViewController : UIViewController
< UIAlertViewDelegate > { //this class conforms to the
UIAlertViewDelegate protocol
}
```

You can even conform to more than one delegate by separating the protocols with commas, like <UIAlertViewDelegate, UITableViewDataSource>.

Once the class conforms to a protocol, you can implement the method in your class without any problem:

```
- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex {
NSLog([NSString stringWithFormat:@"%d", buttonIndex]);
}
```

Delegate

A delegate is just another object that has been assigned by an object as the object responsible for handling events. In the UIAlertView example that we discussed before:

```
UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:@"Hello"
message:@"This is an alert view"
```

```
delegate:self
cancelButtonTitle:@"OK"
otherButtonTitles:nil];
```

The parameter included in the initialiser of the UIAlertView class is called the delegate. If you set this parameter to self, the current object is responsible for handling all the events fired by this instance of the UIAlertView class. If you are not going to handle events fired by this instance at all, you can simply set it to nil:

```
UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:@"Hello"
message:@"This is an alert view"
delegate:nil
cancelButtonTitle:@"OK"
otherButtonTitles:nil];
```

With multiple buttons on the alert view, we were handling the methods defined in the UIAlertViewDelegate protocol to know which button was tapped. This can be either done by implementing it in the same class in which the UIAlertView class was instantiated or by creating a new class to implement the method like:

```
//NewClass.m
@implementation NewClass
- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex {
NSLog([NSString stringWithFormat:@"%d", buttonIndex]);
}
@end
```

You also need to create an instance of this NewClass set it as the delegate to ensure that the alert view knows where to look for the method:

```
NewClass *myDelegate = [[NewClass alloc] init];
UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:@"Hello"
message:@"This is an alert view"
delegate:myDelegate
cancelButtonTitle:@"OK"
otherButtonTitles:@"Option 1", @"Option 2", nil];
[alert show];
```

3 Model-View-Controller (MVC)

Model-View-Controller or the MVC is a common paradigm used to logically divide the code that builds up your GUI (Graphical User Interface). The MVC model of Cocoa Touch divides up the functionality into Model: The classes or objects that hold your data; View: The front panel of your code made up of windows, the area where you user actually interacts with your code; and Controller: Binds the model and view together and is the application logic that decides how to handle the user's inputs. Controller though, can be customised. They are typically subclasses of one of the many generic controller classes from the UIKit framework. An example of this is the UIViewController.

MVC methodology also emphasises strongly upon the reuse of code, to minimise human efforts.

If you again focus on the "Groups and Files" pane in your Xcode window you will notice that it contains of four set of files (let us assume your project is called digit):

```
digitAppDelegate.h
digitAppDelegate.m
digitViewController.h
digitViewController.m
```

The .h files are the header files and .m files are the main source code files. As you would have easily guessed, the third and the fourth files are the ones controlling the ViewController. Let's look at some of the lines of code in our introductory program.

```
#import   <  UIKit/UIKit.h  >
@interface digitViewController : UIViewController {
}
-(IBAction) bnClicked:(id)sender; //Ignore this
@end
```

The first two lines declare your ViewController to be a subclass of the genereic UI View Controller that is available in the UIKit/UIkit.h, and gives us basic functionality in a ready to go manner.

`@end` is simply a marker defining the end of the code.

Till now, we have already seen how to modify any attribute of any object in your view window by using the inspector functionality via GUI. If you want to do the same with code, you will need to create an "outlet". An outlet

mimics the functionality of a pointer that points to an object within in your nib (xib) file. Then, of course, are the actions which we had elaborated on earlier.

```
-(IBAction) bnClicked:(id)sender;
```

IBAction is a special reserved keyword. This also implies a return type of void for your method named `bnClicked` (this can be anything you want).

The (id)sender is the argument which tells your code which was the origin for the stimulus (a tap, flick etc) for your action to be executed, if you want it to be executed irrespective of who triggers it you can also just declare it as:

```
-(IBAction) bnClicked;
```

One important point of note here is that in Objective-C 2.0, the following statements are identical, this will bring in some familiarity to those who are used to Java/C++ "." Notation.

```
myVar = [arbitObject property];
myVar = arbitObject.property
similarly
arbitObject.property = myVar;
arbitObject setProperty:myVar];
```

3.1 Creating an outlet

To create an outlet, select the "File's Owner" item in your digit.xib file and open its Identity Inspector. Below, where you created the `bnClicked:` action, there is an option to create a Class Outlet, by double clicking the + button. Add an outlet called `nmTxtField` and the Type should be set to `UITextField`.

To differentiate further between actions and outlets, an action is a method to handle events invoked by the views (till now we have been dealing with only single view applications) while an outlet allows your code to programmatically modify or reference a view on the View window.

Now again select the `File's Owner` and choose `File > Write Class File` item from the `menu`. This is done to auto generate

Typical events for a text field

the code for what you have done in the last two steps, in the pop-up that follows choose merge. This will open a "diff" like window for you where the left part indicates the code generated by Interface Builder for us and the right indicates the pre-existing code.

The DigitViewController.h file should appear as follows

```
#import  <  UIKit/UIKit.h  >
@interface DigitViewController : UIViewController {
IBOutlet UITextField *nmTextField; // This is the
additional
// line of code corresponding to the outlet you just
created.
// Remember IBOutlet is a reserved keyword
}
- (IBAction)btnClicked:(id)sender;
@end
```

We have demonstrated above how to add outlets via the GUI, to do the same thing with code, you can just append a line in the framework generated above. So if you want to add another outlet by the name of lastNmTextField your digitViewController.h will appear as follows:

```
#import  <  UIKit/UIKit.h  >
@interface BasicUIViewController : UIViewController {
     IBOutlet UITextField *nameTextField;
       IBOutlet UITextField *ageTextField;
}
    - (IBAction)btnClicked:(id)sender;
      @end
```

Now if you check via the GUI by opening the Inspector window for "File's Owner" you would see your newly added outlet

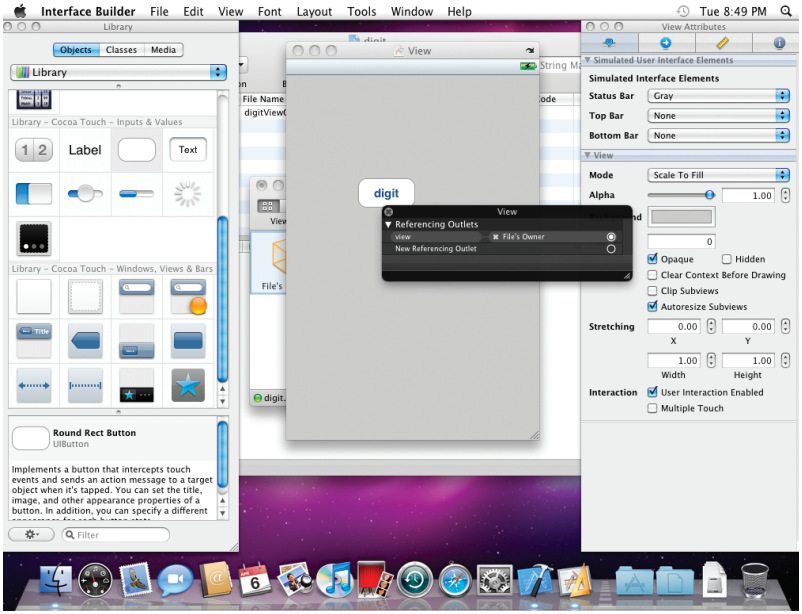
Now refer to an alternate version of the code we had just written

```
#import  <  UIKit/UIKit.h  >
@interface digitViewController : UIViewController {
@property (nonatomic, retain) IBOutlet UITextField
*nameTextField;

}
- (IBAction)btnClicked:(id)sender;
    @end
```

Here the member variable has been explicitly exposed as a property using the `@property` identifier. This is used in combination with the `@`

`sysnthesize` declaration that takes care of declaring the accessor and mutator methods (used to send and receive values to each class instance).



Outlets let you change the behaviour and appearance of your UI via code

3.2 Connecting View Controller to outlets and Actions

Now that you know how to declare actions and outlets both via code and via GUI let us progress towards in detail as to how to link them with your views in the View Window. One method as discussed briefly at the end of chapter 2 is to simply control-click the required object from the view window and drag it to the file's owner icon in the in `digitViewController.xib` window. On realising the mouse a pop-up appears asking you to select an action, where you can select the `bnClicked:` action.

Do the reverse for linking the outlets defined in the View Controller onto the view window, Control-Click the "file's owner" item and drag it to the visual object (say the text field) and drop it there, again a pop-up will appear with a list of outlets that you have defined till now, select the `nmTextField` from there. To enumerate a list of actions and outlet, you can right click the "file's owner" icon. By default whenever you link an action with a button,

the event of a button press, called the `Touched Up Inside` event is linked. This can be modified to any event you like by going to the view window and right clicking on the button, selecting the event and then dragging it to the File's Owner Item. Now that we know how to link outlets and action to viewcontrollers and view, let us get back to our code.

Assuming that you have exposed the outlet as a property as explained above we need to define the getters and setters for this property.

For this open your digitViewController.m file and mention the following:

```
#import  "digitViewController.h"
@implementation digitViewController
@sysnthesize nmTextField
```

We have already connected the "touch up inside" event of the Round Rect Button view with the action bnClick: which was defined in the View Controller. This would be as you might have guessed by now in the `DigitViewController.m` file:

Now refer to an alternate version of the code we had just written

```
#import "digitViewController.h"
@implementation digitViewController
@synthesize nmTextField;
- (IBAction)bnClicked:(id)sender {
      NSString *str = [[NSString alloc] //Static String
Class
         initWithFormat:@"Hello, %@", nmTextField.text ]
     UIAlertView *alert=[[UIAlertView alloc]
                        initWithTitle:@"Your Text below"
                           message: str
                           delegate:self
                           cancelButtonTitle:@"awesome"
                           otherButtonTitles:nil, nil];
     [alert show];
     [alert release];
     [str release];
}
- (void)dealloc {
       [nmTextField release];
     [super dealloc];
}
@end
```

Now if you build and run your application and click on the button the text in the text field will be displayed in the alert window with a cancel button called awesome.

Here, you might want to experiment with adding a "Web view" in your view window, this is a ready to deploy web browser (bare bone) in your application.

Modifying your digitViewController.h and digitViewController.m as below should make your app able to render a simple webpage.

```
#import   <  UIKit/UIKit.h  >
@interface digitViewController : UIViewController {
       IBOutlet UIWebView *webView;
}
@property (retain, nonatomic) UIWebView *webView;
@end
```

As for the .m file do the following

```
#import "digitViewController.h"
@implementation digitViewController
@synthesize webView; //name of your web view
- (void)viewDidLoad { // initiate the loading of the url
//declare a string called strUrl with its content as the URL
      NSString *strUrl = @"http://www.thinkdigit.com";
// Create a URL object with the address from the strUrl string created above
      NSURL *url = [NSURL URLWithString:strUrl];
            NSURLRequest   *request   =   [NSURLRequest requestWithURL:url];
      [webView loadRequest:request];
      [super viewDidLoad];
}
- (void)dealloc {
        [webView release];
      [super dealloc];
}
@end
```

3.3 Creating View controller

Till now we used the "View-Based" application project which already

creates the View Controller configured. This time select a "Window-based Application" project. Let us call it FT.

In a window-based application project no view controller is defined for you. This can be verified by looking at the files in the "Groups & Files" pane.

So now your first step is to add a View Control. To do this, right-click the project name in Xcode and choose Add > New File. In the New File window, under the iPhone OS section appearing on your left, then select Cocoa Touch Class and then select the UIViewController subclass template. You should also tick the "With XIB for user interface" option so that the UI file is also created for you. Click Next to proceed further.

You will be presented with the option of naming your new View Controller class, let us call it NewVC, which will lead to creation of the following files

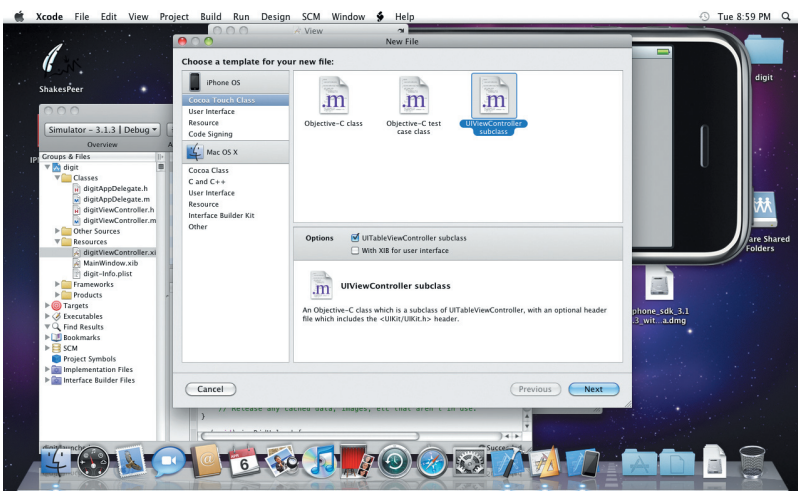
`NewVC.h`

`NewVC.m`

`NewVC.xib`

Now you would need to tell your application that you have created a NewVC view controller, so for this in your Xcode window open up the `FT.h` file and create and instance for NewVC by doing the following

```
#import   <  UIKit/UIKit.h  >
@class NewVC; // This is a forward declaration for the compiler
```



Manually add a view controller

```
  @interface FTAppDelegate : NSObject <
UIApplicationDelegate > {
      UIWindow *window;
        NewVC *NewVC;
  }
  @property (nonatomic, retain) IBOutlet UIWindow *window;
  @property (nonatomic, retain) NewVC *NewVC;
  @end
```

Now move to the FTDelegate.m file, and insert the following code in order to create an instance of the NewVC class and set the view to the current window.

```
  #import "FTAppDelegate.h" // imports the header file
declared above
  #import "NewVC.h"

  @implementation FTAppDelegate
  @synthesize window;
  @synthesize NewVC;
  - (void)applicationDidFinishLaunching:(UIApplication *)
application {
       //The code below is the actual code which creates
the instance for the NewVC class
      NewVC *viewController = [[NewVC alloc]
                               initWithNibName:@"NewVC"
                                       bundle:[NSBundle
mainBundle]];
      //-------------------------------------------------
-----------------------------------
      self.NewVC = viewController;
      [viewController release];

     // adding the physical "View" to the viewcontroller.
      [window addSubview:[NewVC view]];

      [window makeKeyAndVisible];
  }

  //Below is an exhibit in good programming practices.
```

```
- (void)dealloc {
        [NewVC release];
    [window release];
    [super dealloc];
}
 @end
```

Now you can build and run your app and blank view will be displayed in your iPhone Simulator indicating that you have successfully added a view controller manually in an application.

You can again open the NewVC.xib file and customize it the way you want. From this step onward you can work the same way as before to create any application.

4 View

All your GUI controls are displayed here

At the risk of repeating ourselves, every object that you add in the "View" window is a "view", so "Web View" is a view and so is "Textfield", "Label" and even the "Round Rect Button" that you have used till now in the past three chapters. We will now explore some more in a little more details after all GUIs are the main stay of any iPhone application and as demonstrated by the "Web View" above, they are pretty powerful too.

Focus on the "Library" window in your interface builder. Interface builder can be launched by clicking on the xib/nib file.

As shown in the picture above your Library is divided into sections, some of them being:

1) Controllers
2) Data Views (For display for images, tables, data etc)
3) Inputs and Values (Your Round Rect Button and Text field lie here)
4) Windows, Views and Bars (Search Bar, Tool bars and like)

If you notice carefully you will see the absence of one particular view that we have also used previously, the "UIAlertView". This is because it can be only created programmatically as explained in any of the examples. More specifically,

```
UIAlertView *alert = [[UIAlertView alloc]
                      initWithTitle:@"Hello"
                            message:@"This is an
alert view"
                           delegate:self
                           cancelButtonTitle:@"OK"
```

```
                                    otherButtonTitles:@"Option
1", nil];
```

In the above example, instead of just one button till now used for "cancel" function or the exit "function" there will also be another button called the Option 1displayed above the "Ok" button.

4.1 Page control and image view

Notice your iPhone simulator or the iPhone closely, on the menu page there are a series of dots displayed with one of them being lighted up indicating the current page. This is the Page Control view. Now we will mix it with Image View and learn how to create a flappable image app. Let's call this app flip.

So now under the resources tab within your "Groups and Files" pane add 4 images, say by the name of 1.jpg, 2.jpg, 3.jpg and 4.jpg. Open your flipViewController.xib file. Drag and drop two image view from the library (They will be in the in data views section) onto the "View Window", expand them to cover as much area as you want, you can let them overlap, but still they should be arranged in such a way that you should be able to select any one of them individually. We have added two image views to support the flipping animations.

Also, drop in two more image views at this point. They will be used for the temp storage and background image. Select one of the image view and open the Attribute inspector from the tools menu and set the property "Tag" to 0. Repeat the procedure to do the same for the second image view by giving it the "tag" value of 1. Next, drop the "Page Control" view into the "View" window and set its number of pages to 4. The number of pages corresponds to the number of images you want to display.

Open your Xcode and edit the flipViewController.h file to define the following outlets.

```
#import <UIKit/UIKit.h>
@interface flipViewController : UIViewController {
     IBOutlet UIPageControl *pageControl;
     IBOutlet UIImageView *imageView1;
     IBOutlet UIImageView *imageView2;
    UIImageView *tempImageView, *bgImageView; // For the
two additional image views that you added.
}
@property (nonatomic, retain) UIPageControl *pageControl;
@property (nonatomic, retain) UIImageView *imageView1;
```

```
@property (nonatomic, retain) UIImageView *imageView2;
@end
```

Now move to your flipViewController.m file and edit it as follows:

```
#import "flipViewController.h"
@implementation flipViewController
@synthesize pageControl;
@synthesize imageView1, imageView2;
 (void) viewDidLoad {   //Void is the return type
   //Set the first image in imageView1 and load the 2nd
in the tempImage view
        [imageView1  setImage:[UIImage  imageNamed:@"1.
jpeg"]];
     tempImageView = imageView2;
     [imageView1 setHidden:NO];
     [imageView2 setHidden:YES];  //Since we don't want
to display the 2nd image yet
// This following declares the event hander for the page
control
              [pageControl   addTarget:self   action:@
selector(pageTurning:)
         forControlEvents:UIControlEventValueChanged];
     [super viewDidLoad];
}
- (void) pageTurning: (UIPageControl *) pageController
// To account for a change of page
// Void is the return type for this method
{
//Declare and interger by the name of next page of type
NSInteger
// Load the value of the current page in it
     NSInteger nextPage = [pageController currentPage];
// A very c/c++ similar switch statement using the
"nextPage" variable
     switch (nextPage) {
         case 0:
                        [tempImageView  setImage:[UIImage
imageNamed:@"1.jpeg"]];
              break;
```

```
            case 1:
                            [tempImageView  setImage:[UIImage
imageNamed:@"2.jpeg"]];
                 break;
            case 2:
                            [tempImageView  setImage:[UIImage
imageNamed:@"3.jpeg"]];
                 break;
            case 3:
                            [tempImageView  setImage:[UIImage
imageNamed:@"4.jpeg"]];
                 break;
            default:
                 break;
       }
       // actual flip between the two images
       if (tempImageView.tag==0) {
            tempImageView = imageView2;
            bgImageView = imageView1;
       }
       else {
            tempImageView = imageView1;
            bgImageView = imageView2;
       }
       // Following is the code for animation. Most
     [UIView beginAnimations:@"flipping view" context:nil];
       [UIView setAnimationDuration:0.5];
        [UIView setAnimationCurve:UIViewAnimationCurveEase
InOut];
                    [UIView    setAnimationTransition:
UIViewAnimationTransitionFlipFromLeft
            forView:tempImageView cache:YES];
       [tempImageView setHidden:YES];
       [UIView commitAnimations];
          [UIView  beginAnimations:@"fl      ipping  view"
context:nil];
      [UIView setAnimationDuration:0.4]; //Time in seconds
       [UIView setAnimationCurve:UIViewAnimationCurveEase
```

```
InOut];
                    [UIView    setAnimationTransition:
UIViewAnimationTransitionFlipFromRight
           forView:bgImageView cache:YES];
      [bgImageView setHidden:NO];
      [UIView commitAnimations];
  }
  //Your main code is ended what follows is the clean up.
  - (void) dealloc {
      [pageControl release];
      [imageView1 release];
      [imageView2 release];
      [super dealloc];
  }
  - (void) didReceiveMemoryWarning {
           [super didReceiveMemoryWarning];
  }
  - (void) viewDidUnload {
  }
  @end
```

You can now build and run by pressing [command] + [R], to simulate your application on the iPhone simulator, tapping the page control at the bottom should flip the image displayed above.

4.2 Segmented Control

If you find the mammoth real estate provided to you by the iPhone a little limiting still, panic not, iPhone has its own implementation of the tabbed style interface called the "Segmented Control

This time create a new View-Based Application project and call it SegC. Open the SegCViewController.xib to add views to it via the interface builder.

From the library drag the following views into your "View" window:

1) Segmented Control
2) View
3) Round Rect Button

When you drop the second view onto your "view" window, ensure you don't end up making the second view a child of the first one, this can be done by shifting from icon view to list view in the SegCViewController.xib window and ensuring that the both the "View" of type " UIView" are at the

same hierarchy.

Name the two Round Rect Buttons as "Button 1" and "Button 2" from their respective inspector window, and put them in View1 and View 2.

Modify your SegCViewController.h file to add the following outlets:

```
#import <UIKit/UIKit.h>
@interface SegCViewController
: UIViewController {
      IBOutlet UIView *view1;
      IBOutlet UIView *view2;
     IBOutlet UISegmentedControl
*segmentedControl;
}
@property (nonatomic, retain)
UIView *view1;
@property (nonatomic, retain)
UIView *view2;
@property (nonatomic, retain) UISegmentedControl
*segmentedControl;
@end
```

IPhone's way of tabbed interface

Now you need to connect the declared outlets above to their respective views. This can be done graphically via the Interface Builder. After you have connected them, it is time to implement the action, for which open Xcode and edit the SegCViewController.m

```
#import "SegCViewController.h"
@implementation SegCViewController
@synthesize segmentedControl;
@synthesize view1, view2;
- (void) viewDidLoad {
     //The event handler for the segment control, by the
name of segmentChanged
          [segmentedControl addTarget:self action:@
selector(segmentChanged:)
      forControlEvents:UIControlEventValueChanged];
      [super viewDidLoad];
}
```

```
  // This is the declaration of an action which accounts
for a change in the segmented control
  -  (IBAction) segmentChanged:(id)sender {
  // Loads the selectedSegmentIndex property of the
SegmentedControl into a ineger variable
  // called the selectedSegment
        NSInteger selectedSegment = segmentedControl.
selectedSegmentIndex;
  // Now you can either use a switch case statement as
demonstrated in the last example
  // But since we are switching only between two different
views so you can use an if-else ladder
      if (selectedSegment == 0) {
           [self.view1 setHidden:NO];  //Equal to making
the view1 visible
           [self.view2 setHidden:YES];
      }
      else{
           [self.view1 setHidden:YES];
           [self.view2 setHidden:NO];
      }
  }
  -  (void) dealloc {
      [segmentedControl release];
      [view1 release];
      [view2 release];
      [super dealloc];
  }
```

Done. Yes, it's that simple. Build and Run your application to test it in the iPhone simulator. In the simulator you can change the view in focus by tapping on the selector bar at the top.

4.3 Multiple Views using Buttons

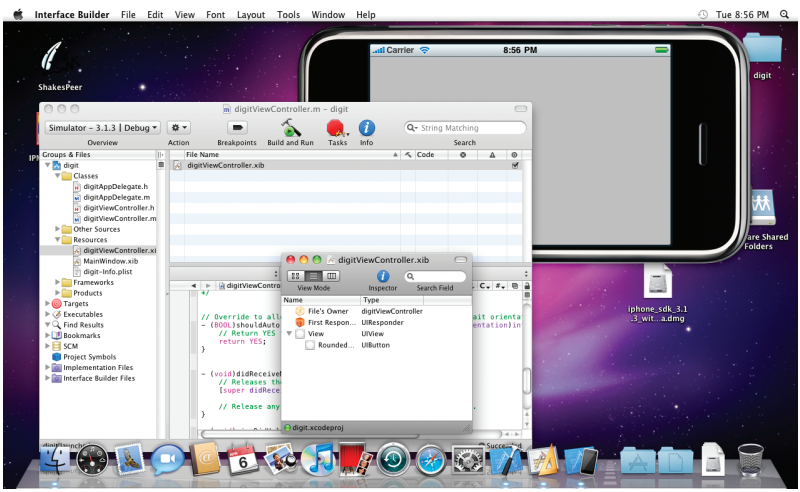
There may be cases when you want multiple views like the above but don't want them to be displayed all at the same time, or want them to be displayed based on the User Interaction. Consider an example where your main view displays two buttons called "Choice1" and "Choice2". Based on which button your user clicks he is displayed a specific view which has another

button called “main menu” which takes him back to the original main menu view. To establish the above, start by creating a “View-Based Application” in Xcode and call it “Mview” for multiple views. As always open the MviewViewController.xib file in the interface builder. Add two “button views” to it and call them choice 1 and choice 2. Switch to Xcode and modify your MviewViewController.h as follows:

```
#import <UIKit/UIKit.h>
@interface MviewViewController : UIViewController {
}
//Following defines the action for both buttons.
-(IBAction) displayView1:(id) sender;
-(IBAction) displayView2:(id) sender;
@end
```

Back in Interface builder link the “Choice 1” and “Choice 2” button to the “File’s Owner” by dragging them. This will by default link the “touch up inside” event of the two buttons to the displayView: action created above.

Now in Xcode, right-click the classes group and add a new file. Choose the “UIViewController” subclass template. Here you will need to add the xib file explicitly since there is no option to auto generate it. Xib file can be added by right-clicking the resources group and then under the “User Interfaces” Group.



Available in the User Interface Group

Repeat the above procedure twice and call the two new views as oneViewController and twoViewController respectively and name the xib as oneview.xib and twoview.xib. Now open the oneview.xib and twoview.xib in the interface builder and select their respective "File's Owner" item and open the identity Inspector. Set their class to oneViewController and twoViewController. Within each of the oneview.xib and twoview.xib you should connect the File's Owner of each with its own view. Open up each of the two "view" windows of the just created new views and add a "menu" button to it. Let us call it bnMenu. In Xcode modify the two files as following

OneViewController.h

```
#import <UIKit/UIKit.h>
@interface oneViewController : UIViewCont
}
-(IBAction) btnMenu:(id) sender;
@end
TwoViewController.h
#import <UIKit/UIKit.h>
@interface twoViewController : UIViewCont
}
-(IBAction) btnMenu:(id) sender;
@end
```

You now need to connect the menu button in each view with its respective File' Owner item, this can be done via the interface builder.

Add the following code to the respective .m files, so that on clicking the return button it is removed from the view

OneViewController.m

```
#import "oneViewController.h"
@implementation oneViewController
-(IBAction) btnMenu:(id) sender {
    [self.view removeFromSuperview];
}
@end
```

TwoViewController.m

```
#import "TwoViewController.h"
@implementation oneViewController
-  (IBAction) btnMenu:(id) sender {
    [self.view removeFromSuperview];
}
```

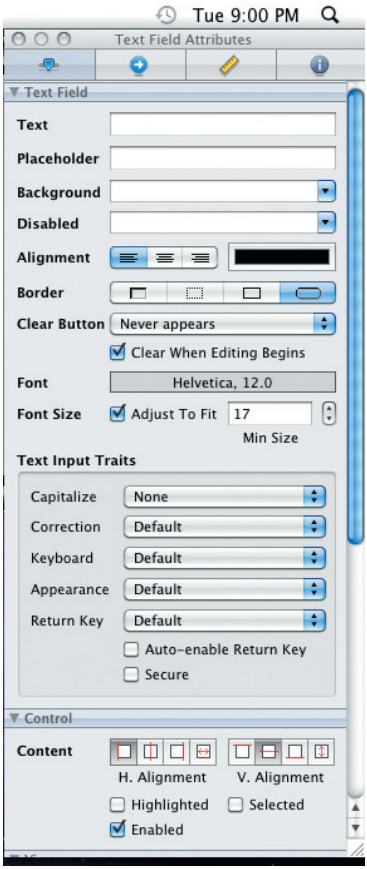
```
@end
```

Finally, as the last step add the code to your main view .m file. Open the MviewViewController.m and modify it as follows

```
#import "MviewViewController.h"
#import "oneViewController.h"
#import "twoViewController.h"
@implementation MviewViewController
oneViewController *oneViewController;
twoViewController *twoViewController;
// The following two methods add the two views on the
click of the
// respective choice button onto the mainview.
-(IBAction) displayView1:(id) sender{
    oneViewController = [[oneViewController alloc]
    initWithNibName:@"oneView" bundle:nil];
    [self.view addSubview:oneViewController.view];
}
-(IBAction) displayView2:(id) sender{
    twoViewController = [[twoViewController alloc]
    initWithNibName:@"twoView" bundle:nil];
    [self.view addSubview:twoViewController.view];
}
- (void)dealloc {
    [oneViewController release]
    [twoViewController release];
    [super dealloc];
}
@end
```

5 Keyboard

iPhone is one of the few mobile devices that can display a usable QWERTY keypad in both the portrait and landscape orientation, courtesy its heavy width. The iPhone keyboard is also intelligent in the sense that it can automatically correct common spelling mistakes and like. In this chapter, we will learn to deal with keyboard, its variant and orientation. As a developer for the iPhone platform, you should be able to successfully summon the keyboard and make it disappear when you don't need it.



You configure the handling of text

5.1 Input types

The keyboard is displayed by default, when you click any text field view. Now we will delve into the topic in further detail. To begin, open up Xcode and create a new View-Based application, let us name it kb. Open the auto generated kbViewController.xib file in the interface Builder. Add onto the view a Label and a Text Field.

Open the `tools > attribute inspector` for the TextField View that you have just added.

Pay special attention to the "Text Input Traits", which gives you the following options

Capitalize: Auto capitalize the text that is entered, useful if you are requesting an alphanumeric serial in which all the alphabets are by default capitalised.

Correction: You can set it according to your wish. This will enable whether the keypad also gives suggestion regarding spelling to the user. So

say in a field where you are accepting proper nouns (especially in India). It would be a good idea to switch it off.

Keyboard: There are three major types available apart from default, the email address, Phone Pad, and Number Pad. So in case you are asking for a numeric pin code, it would be advisable to switch to the numeric pad while entering a phone number. It is recommended to switch to the Phone Pad option.

Appearance: Basic appearance characteristics

Return Key: Modify the type and behaviour of your return key. There are lots of options available here like Go, Google, Next, Join, Route, Search, Send, Yahoo and Done.

5.2 Make it vanish

The KeyBoard is auto summoned when any TextField View gets activated. But what if (and this is a common scenario) you want the keypad to go away after a user hits the return key or any of its variation while the TextField is still active. Consider the following: you have asked the user for a search term and set your return key as "search". Now when the user hits the search button after typing his/her query, the TextField view is still active, but ideally the keypad should disappear. This is exactly what we will learn in this section.

We will continue with the KB application that you had created above. Select the "File's Owner" icon in the kbViewController.xib window and view its Identity Inspector window. Over here, add a new action by clicking on the "+" icon below the Class Actions section. Let us call it "vanish".

Now switch to your "View" window and right-click the TextField view in it. This will display the list of possible events in it. Select "Did End on Exit" and then drag it to the "File's Owner" item. This should give you the pop-up with the "vanish" action that you have just created, select it. This should be enough for the kbViewController.xib file, save and exit it.

Back to your Xcode, modify your kbViewController.h file as follows:

```
#import <UIKit/UIKit.h>
@interface kbViewController : UIViewController {
}
-    (IBAction) vanish:(id) sender;
@end
```

Now that you have edited your header file, open your xbViewController.m file:

```
#import "KbViewController.h"
```

```
@implementation KbViewController
-(IBAction) vanish:(id) sender{
    [sender resignFirstResponder];
}
```

Now save, build and run your project. This will launch the application in the simulator. As expected, the keyboard will appear when you click on the "TextField" view and disappear as you press the return key. So far so good, but what if there is no return key as in the case of Number Pad? The first step is to set the keyboard type as numeric pad. To do this, open the kbViewController.xib file in the interface builder, then select the TextField View and open its Attribute inspector from the tools menu. Under the text input traits, change the keyboard type to Number Pad. Now select the "File's Owner" item in the kbViewController.xib window and view its identity

Number Pad layout of the iPhone

Numbers and Punctuation Layout

inspector. Add an action over called backTouch and also add an outlet. Let us call the outlet TextField only. Now we need to link this outlet so Control-drag the File's Owner item from the kbViewController.xib window onto the TextField view in the "view" window. The textField outlet you just created should pop-up, select it.

Next add a button to your "view", or as Apple calls it a "Round Rect Button View" to your "view" window. With the Round Rect Button view selected, choose layout-> Send back. Now resize the button so it covers your entire view that is either 460x320 pixel if you are allowing for the status bar or 480x320 pixels if you want full screen. Then open the inspect window of the Round Rect Button view and set its type to Custom.

Now you need to link the Round Rect button view with its associated action, so control-drag it onto the file's owner item in the kbViewController.xib window. Select the backTouch action from the pop-up menu. You are now done with the xib file, save and exit it.

In your Xcode window start with editing the kbViewController.h file and modify it as follows:

```
#import <UIKit/UIKit.h>
@interface KbViewController : UIViewControl
 IBOutlet UITextField *textField;
}
@property (nonatomic, retain) UITextField *textField;
- (IBAction) backTouch:(id) sender;
- (IBAction) vanish:(id) sender;
@end
```

Now moving to the kbViewController.m file

```
#import "kbViewController.h"
@implementation kbViewController
@synthesize textField;
//This is the action details for your invisible //button
cover the whole screen.
//This explicitly tells the textField to loose focus
- (IBAction) backTouch:(id) sender{
    [textField resignFirstResponder];
}
- (IBAction) vanish:(id) sender{
    [sender resignFirstResponder];
}
```

That is it, now follow the standard routine of saving, building, running and enjoying your new app in the simulator. Touching anywhere outside the textfield view will activtate the touch up inside action of the round rect button view and make the keyboard disappear.

5.3 Relocating views

We have discussed in detail about how to make the keyboard disappear when your user is done editing, but there is one more problem, when the keyboard is on the screen, it occupies a good 216 pixels in height. To detect the presence of the keyboard and manually relocate stuff as per your needs, your code should be aware of the following two notifications:

1)UIKeyboardWillShowNotification and UIKeyboardWillHideNotification

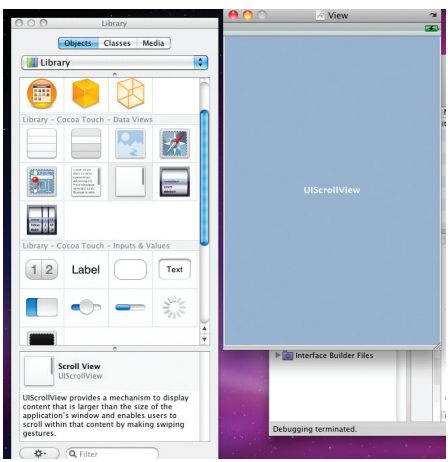
2) textFieldDidBeginEditing and textFieldDidEndEditing

The first is used to detect the keyboard's presence and the second is used to detect which text field is being edited so that you can relocate.

We will start by creating a new View-Based application that we'll call funky. Open your funkyViewController.xib (see, it seems fun) in the interface builder and add a ScrollView view to your "view" window. Now add a TextField. This would be needed to activate the keyboard and a Round Rect Button View onto your ScrollView area. The ScrollView will be used to scroll up other views when the keyboard appears on screen.

We need to create two outlets to interact with scrollView view and the TextField Field. Let us intuitively name them as scrollView and textField itself. Follow the usual process of creating them via the Identity Inspector window of the file's owner item in the funkyViewController.xib and then clicking on the "+" button in the Class Outlets section

To link it, control-drag the File's Owner item to the ScrollView view and then select the scrollView outlet that you



Scroll View will be used for avoiding the keyboard

have just created. Next Control-drag the TextField view onto the File's Owner item.

Save and exit your funkyViewController.xib and open the funkyViewController.h in Xcode. Modify it as following:

```
#import <UIKit/UIKit.h>
@interface funkyViewController :
    UIViewController <UITextFieldDelegate> {
    IBOutlet UITextField *textField;
    IBOutlet UIScrollView *scrollView;
}
@property (nonatomic, retain) UITextField *textField;
@property (nonatomic, retain) UIScrollView *scrollView;
@end
```

Gear up now to do a bit of coding. It is long but simple to grasp. Modify your funkyViewController.m file as following

```
#import "funkyViewController.h"
@implementation funkyViewController
@synthesize textField;
@synthesize scrollView;
//Below is the declaration for three variables each //
storing the size of the keyboard, size of the //screen,
and to save the original state of screen to //revert back
later. The size of keyboard and screen //can be hardcoded
but it is advisable to obtain it //dynamically during
runtime to accommodate for //changes.
CGRect keyboardBounds;
CGRect applicationFrame;
CGSize scrollViewOriginalSize; //=scrollView.contentsize
in viewDidLoad

// Below is just simple arithmetic to calculate the //
center positions and the amount of pixels needed to //
scroll
-(void) moveScrollView:(UIView *) theView {
        CGFloat viewCenterY = theView.center.y;

        CGFloat freeSpaceHeight = applicationFrame.size.
height -
```

```
                                    keyboardBounds.size.height;

                    CGFloat scrollAmount = viewCenterY -
freeSpaceHeight / 2.0;

       if (scrollAmount < 0)  scrollAmount = 0;

         scrollView.contentSize = CGSizeMake(
                                       applicationFrame.size.
width,
                                       applicationFrame.size.
height +
                                         keyboardBounds.size.
height);

               [scrollView setContentOffset:CGPointMake(0,
scrollAmount) animated:YES];
  }
  //This notifies your application which textField if //
multiple are being activated and calls the //moveScrollView
method to begin the scrolling.
  -(void)   textFieldDidBeginEditing:(UITextField   *)
textFieldView {
       [self moveScrollView:textFieldView];
  }
  //Animations are used to restore the original state //of
the view items. If this is not done once the //editing is
done there will be an abrupt change on //screen.
  -(void)    textFieldDidEndEditing:(UITextField    *)
textFieldView {
        [UIView beginAnimations:@"back to original size"
context:nil];
       scrollView.contentSize = scrollViewOriginalSize;
       [UIView commitAnimations];
  }
  //This is where your keyboard size will be passed on //
the variable that you declared above. This will //happen
when the keyboard will appear on screen
```

```
-(void) keyboardWillShow:(NSNotification *) notification
{
        NSDictionary *userInfo = [notification userInfo];
      NSValue *keyboardValue = [userInfo objectForKey:UI
KeyboardBoundsUser
      [keyboardValue getValue:&keyboardBounds];
}
-(void) keyboardWillHide:(NSNotification *) notification
{
}

//viewWillAppear gets invoked before the view appears
//on screen and does the registration for the two //
notifications described above
-(void) viewWillAppear:(BOOL)animated
{
      [[NSNotificationCenter defaultCenter]
            addObserver:self
            selector:@selector(keyboardWillShow:)
            name:UIKeyboardWillShowNotification
            object:self.view.window];

      [[NSNotificationCenter defaultCenter]
            addObserver:self
            selector:@selector(keyboardWillHide:)
            name:UIKeyboardWillHideNotification
            object:nil];
}
//Simply removing the notifications when the view //
disappears.
-(void) viewWillDisappear:(BOOL)animated
{
      [[NSNotificationCenter defaultCenter]
            removeObserver:self
            name:UIKeyboardWillShowNotification
            object:nil];
      [[NSNotificationCenter defaultCenter]
            removeObserver:self
```

```
            name:UIKeyboardWillHideNotification
            object:nil];
  }
  -(void) viewDidLoad {
      scrollViewOriginalSize = scrollView.contentSize;
           applicationFrame = [[UIScreen mainScreen]
applicationFrame];
      [super viewDidLoad];
  }
  -(BOOL) textFieldShouldReturn:(UITextField *)
textFieldView {
      if (textFieldView == textField){
          [textField resignFirstResponder];
      }
      return NO;
  }
  -(void) dealloc {
      [textField release];
      [scrollView release];
      [super dealloc];
  }
```

Save the project, cross your fingers, debug the code to check for any errors, say a little prayer and then build and run your application. Now when the application is running in the simulator, you will notice that tapping (activating) the text field will scroll up the TextField view, Round Rect button view and the Label view above the keyboard. Mission Accomplished.

6 AutoSizing and AutoRotating

When the iPhone was launched, it was pretty much the first device to effectively support an inbuilt accelerometer that allowed it to adjust to its orientation. It was one of the most talked about feature and for what it is worth it was actually a USP alongside the awesome capacitive touch screen.

We had touched upon this aspect in the earlier chapters but now is the time to explore it further and gain a better functional understanding of how to pretty-face your UI and make it adaptable to both the landscape and portrait orientations.

The importance of this feature can be gauged by the simplicity with which it can be handled by the SDK as compared to the KeyBoard handling. Most of the complicated part, like taking direct readouts from the accelerometer is hidden from the eye of the developer. Even GUI movement is supported by the SDK with its anchoring and resizing features.

Let us make a blank view-based application and call it RotateMe in Xcode. The first step would be to enable orientation feedback. This can be achieved by opening the RotateMeViewController.m in your Classes folder [In the Groups and Files] pane. Uncomment the following codeblock

```
// Override to allow orientations other than the default portrait
// orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```

If you notice the return line above, it returns a True or YES only for the

```
// Override to allow orientations other than the default portrait orienta
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)in
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}


- (void)didReceiveMemoryWarning {
```

→ your key to unlock rotation abilities

UIInterfaceOrientationPortrait, which is the default orientation. Hence, for all practical purpose disable auto-rotation. To enable auto-rotation simply make it return YES, so your code block should now look like

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
interfaceOrientation {
    return YES;
}
```

There are four types of orientation that are supported and recogonized, namely

1) UIInterfaceOrientationPortrait
2) UIInterfaceOrientationPortraitUpsideDown
3) UIInterfaceOrientationLandscapeLeft
4) UIInterfaceOrientationLandscapeRight

6.1 Rotation via Autosize

Open your Resouces folder again in Xcode and double-click on the RotateMeController.xib to open it in the Interface Builder.

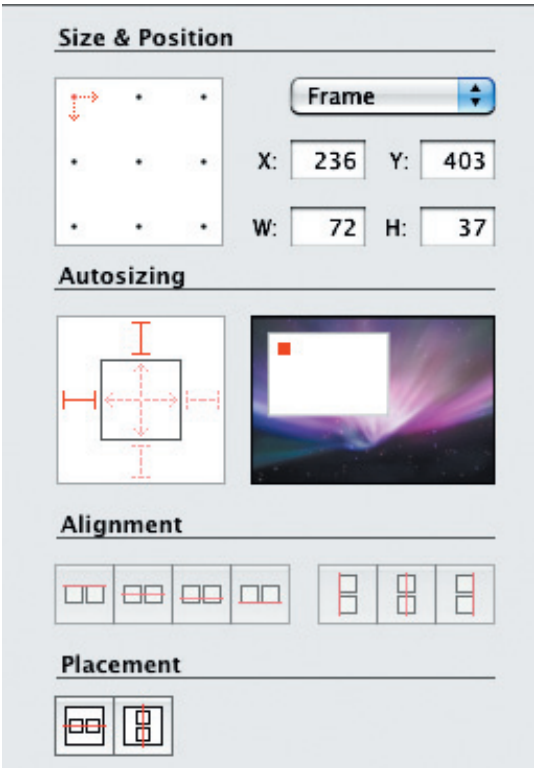
Add a Round Rect button View to the top left of your "View" window and call it "one" and make another one at the bottom right of the "View" window and call it "two". You can edit the title of any button by merely double clicking on it. Build and run your project. In the simulator when you rotate your simulator, you will notice that the Round Rect button View called two has

Rotation without autosize or anchoring attributes

disappeared from your screen, something which is not entirely desirable. Let us go ahead and fix it now.

In the interface builder, bring up the size inspector from the tools menu after selecting the Round Rect Button View named "One".

Within the size inspector, the thing that is of most importance to us right now is the "AutoSizing" section. The section is divided into parts - one being the square within a square part with arrows point outwards, and the other being the preview section (the one on the right). The preview helps save time, by removing the need to build and run the whole application while modifying properties for each individual item.



You can details your positions here

Focussing back now one the left box. It is made of two kinds of arrows/lines. One which are inside the inner square and second which are outside the inner square. The inner square is representative of the object selected, so since right now the size inspector is open for the Round Rect Button View named one, inner square represent that.

The red arrows inside the inner square symbolize the horizontal and vertical space inside the object. Clicking on any arrows toggles its state between solid and dash. A solid arrow signify that the width/height in that direction is free to change as the window resizes and if the arrow is dashed then it represents fixed property.

The arrows outside the inner square represent the distance from the edge of the selected object to the edge of the view that contains it. The view which contains another view object can be referred as the "super view". The solid and dash have the same meaning as for the arrows inside the square.

So now coming back to our problem of displaying the top left button and the bottom right button at the same relative position independent of the orientation of the screen. Since we don't need to resize them as of now we can leave all the inside arrows fixed or effectively dashed.

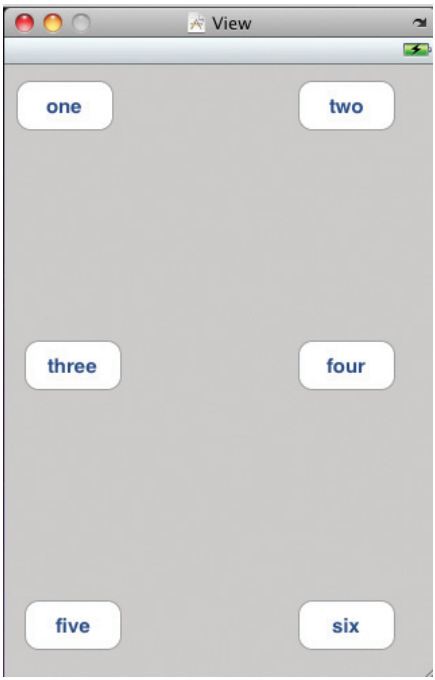
Now as for the outside arrows, for our Round Rect Button View one [at the top left corner] leave the top and left arrows as flexible or solid and similarly for our Round Rect Button View two [at the bottom right] leave the right and bottom outside arrows flexible or solid. Leave everything else as dashed or fixed. Now build and run your application and when the simulator fires up, rotate it (either via the `[command]` + right arrow combination or via rotate right or rotate left from the hardware menu). This time you will notice the desired result and everything will be visible on the screen with Round Rect button one always being at top left and the Round Rect Button two always at bottom right.

The final result

If you had kept the size of the buttons to look reasonable in the portrait orientation they might look relatively small in the landscape mode, if you think so, go play around with the inside arrows.

6.2 Moving Objects to accommodate rotation

In the example that follows, you don't move any of the "sub views" inside your "super view" explicitly when the phone is rotated. In a complex GUI layout, the method described earlier may not be the optimal choice. For this there's another way to handle rotation. A typical scenario is where 2x3 configuration of some view object say a Round Rect Button view or TextField in the portrait configuration. In our approach till now, even in landscape mode it will retain the 2x3 configuration instead of the more intuitive 3x2 configuration. This may lead to a wastage of space in the centre and crowding at side. Now we will learn how to fix this.



Six Round Rect button View in 2x3 configuration

Start by creating a blank project of type View-Based. Let us call it MoveIt. Now open your MoveItViewController.xib in interface builder and load it with six Round Rect Button View in a 2x3 orientation. Call them button1, button 2 and so on.

Now our basic problem is to change the attributes of an object on rotation. We already know that to change any attribute programmatically we need to declare an outlet for it.

Open Xcode and start modify the MoveItViewController.h file as following

```
#import <UIKit/UIKit.h>
@interface MoveItViewController : UIViewController {
    UIButton *button1;
    UIButton *button2;
    UIButton *button3;
    UIButton *button4;
    UIButton *button5;
```

```
    UIButton *button6;
}
@property (nonatomic, retain) IBOutlet UIButton *button1;
@property (nonatomic, retain) IBOutlet UIButton *button2;
@property (nonatomic, retain) IBOutlet UIButton *button3;
@property (nonatomic, retain) IBOutlet UIButton *button4;
@property (nonatomic, retain) IBOutlet UIButton *button5;
@property (nonatomic, retain) IBOutlet UIButton *button6;
@end
```

Now you have to link them. For this open the interface builder by double-clicking on the MoveItViewController.xib file and Control-drag from the "File's Owner" icon to each of the six buttons, and select the respective action for each from the pop-up.

Now you need to edit the MoveItViewController.m file to manually override the default rotation policy. For this open the file and edit it as following

```
#import "MoveItViewController.h"
@implementation MoveItViewController
//To auto create accessors and mutators for each //
configuration
@synthesize button1;
@synthesize button2;
@synthesize button3;
@synthesize button4;
@synthesize button5;
@synthesize button6;
//Manual overriding of the default rotation mechanism.
|| //signifies OR, so that in both cases of portrait
orientation //the same behaviour is observed, normal or
upside down.
- (void)willAnimateRotationToInterfaceOrientation:(UIIn
terfaceOrien
          interfaceOrientation duration:(NSTimeInterval)
duration {
                    if (interfaceOrientation ==
UIInterfaceOrientationPortrait
          || interfaceOrientation ==
              UIInterfaceOrientationPortraitUpsideDown) {
```

```
            button1.frame = CGRectMake(20, 20, 125, 125);
            button2.frame = CGRectMake(175, 20, 125, 125);
            button3.frame = CGRectMake(20, 168, 125, 125);
           button4.frame = CGRectMake(175, 168, 125, 125);
            button5.frame = CGRectMake(20, 315, 125, 125);
           button6.frame = CGRectMake(175, 315, 125, 125);
       }
  //CGRectMake is function included in your sdk. The four
//arguments passed onto it are the X,Y position along with
the //width and height of the rectangle that you want to
create.
       else {
            button1.frame = CGRectMake(20, 20, 125, 125);
            button2.frame = CGRectMake(20, 155, 125, 125);
            button3.frame = CGRectMake(177, 20, 125, 125);
           button4.frame = CGRectMake(177, 155, 125, 125);
            button5.frame = CGRectMake(328, 20, 125, 125);
           button6.frame = CGRectMake(328, 155, 125, 125);
       }
  }
  - (BOOL)shouldAutorotateToInterfaceOrientation:
       (UIInterfaceOrientation)interfaceOrientation {
       return (interfaceOrientation !=
            UIInterfaceOrientationPortraitUpsideDown);
  }
  - (void)didReceiveMemoryWarning {
       [super didReceiveMemoryWarning];
  //Prevents memory leaks, clears cache and takes care of
no //availability of super view
  }
  - (void)viewDidUnload {
       self.button1 = nil;
       self.button2 = nil;
       self.button3 = nil;
       self.button4 = nil;
       self.button5 = nil;
       self.button6 = nil;
       [super viewDidUnload];
```

```
}
- (void)dealloc {
    [button1 release];
    [button2 release];
    [button3 release];
    [button4 release];
    [button5 release];
    [button6 release];
    [super dealloc];
}
```

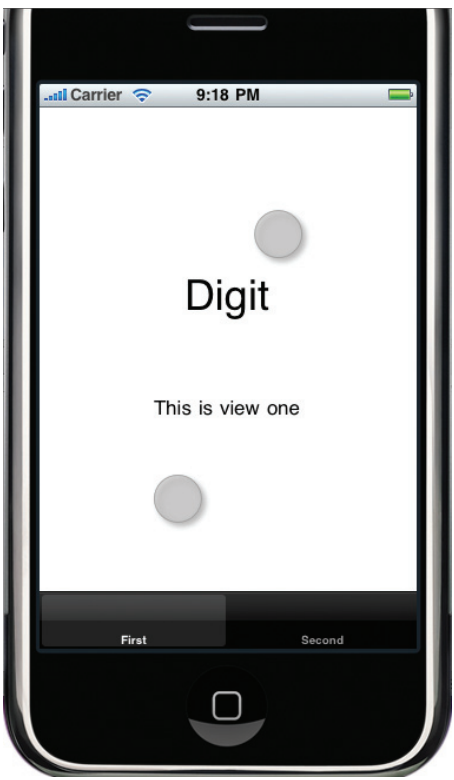
Now you are done, build and run the application, let it launch in the simulator and then on rotation, observe the neat behaviour. There is another way to accommodate for rotation, in that you create two views one for each orientation with a copy of each "view" item in both the views. Each copy is linked to every action and outlet [separate for each copy] performing the same function.

7 Bars, tabs and navigation

7.1 More stuff in 480x320

We have already considered two ways of creating multi-view application and switching between them via either segmented control or Round Rect button View based switch. The problem of limited visual space is encountered so often that the iPhone SDK provides a more polished way of dealing with them. They are referred to as "Tab-bar" Application.

If you have wondered how to create the iPod-style navigation based application, we shall also attempt to answer that question now. An example of this style of application is the settings application in your iPhone or iTouch. Here you are displayed with a list of items and you can select one to be displayed with another sub list. We will discuss creation of both kind of application which will complete your arsenal of designing a GUI meeting the most encountered problems. Let us start with Tab-Bar applications.



A two tab UI displayed in the simulator. Dots reflect multi-touch

7.2 Tab bar applications

Without wasting further time on textual theory, stretch out your fingers and launch XCode, but this time choose Tab-Bar Applications. Let us call it "tba" for Tab Bar Application. We will begin with analysing and understanding what is provided to us by default in the template and then later move on to adding new tabs.

If you notice in your "groups and files" pane, this time for *.xib files there are two files,

namely MainWindow.xib and SecondView.xib. Now open the contents of tbaAppDelegate.h file and you will notice that instead of the standard UIViewController class that you have been using till now has been replaced by the UITabBarController class. It should resemble something like the following

```
#import <UIKit/UIKit.h>
@interface tbaAppDelegate : NSObject
    <UIApplicationDelegate, UITabBarControllerDelegate>
{
    UIWindow *window;
    UITabBarController *tabBarController;
}
@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UITabBarController
*tabBarController;
@end
```

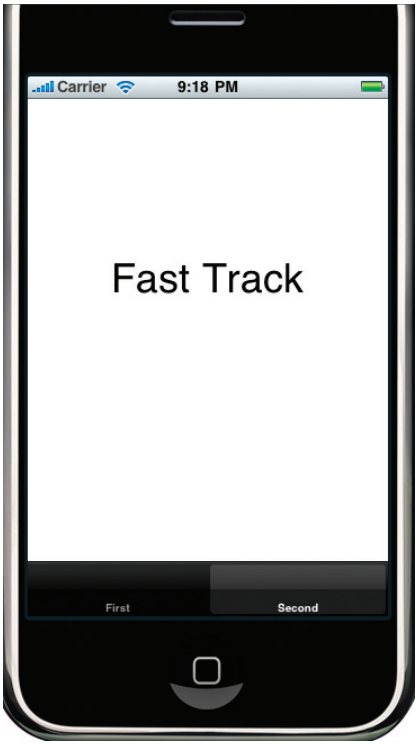
Similarly, an examination of the tbaAppDelegate.m file you will notice the following

```
import "tbaAppDelegate.h"
@implementation tbaAppDelegate
@synthesize window;
@synthesize tabBarController;
- (void)applicationDidFinishLaunching:(UIApplication *)
application {
// You wish to now add the the tab bar controller's
present view as // a subview of the window. This is
achieved by the following:
    [window addSubview:tabBarController.view];
}
```

But the biggest visual difference will come when you will open the MainWindow.xib to edit via the Interface Builder, you will notice that instead of the usual blank "View" window, you will already see a tab bar at the bottom, with First and Second as their default labels. This looks similar to your "phone" application in an iPhone that also has tabs below it for your contacts. If you open the Identify Inspector for the "View" you will notice that it is actually a ViewController and the implementing class is the FirstViewController. Now move your attention towards the windows titled "MainWindow.xib" you will notice in addition to the usual File's Owner

and First Responder Item there is a "Tab Bar Controller" instead of the "View" item. This is where all the UI magic happens. The Tab Bar Controller is a container which has multiple View Collections inside it. By Default it comes loaded it with two of them.

Now that we understand the basics of the "Tab-Bar" Applications the most natural question to ask is how to add more tabs?



the alternate view of your two-tab application

7.3 Adding tab bar items

Open your library window inside the interface builder and drag and drop "Tab Bar Item" on the pre existing Tab Bar view. Now select the Tab Bar item that you have just dragged and drop and open its Attribute Inspector window. From here you can change its title and icon displayed via the "Title" and "Image" attribute. Apple makes your job easier and gives you pre-designed combinations for common tasks. For this you will need to change the identifier setting, for the purpose of illustration let us set it to "Search".

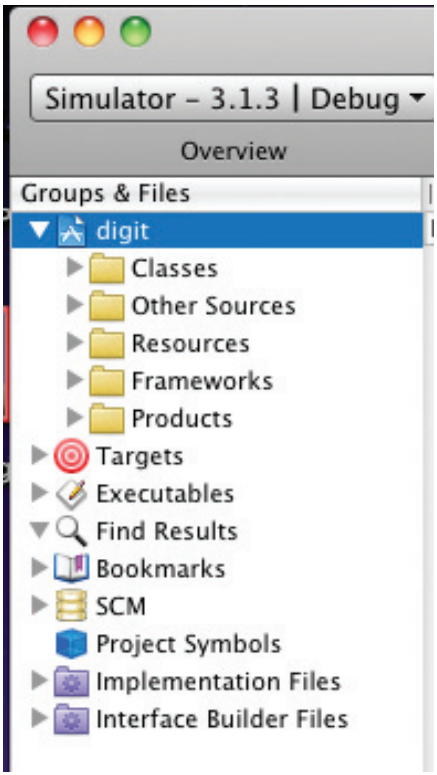
You will immediately notice that your third bar is now by default called "Search" and has a Magnifying glass icon next to it. The badge attribute that you see in the inspector window is super script text badge that appears next to your icons. If you wish fill it with a number like say "3" or you can even leave it blank.

In your interface builder you can open each of the views separately and modify their appearances and behaviour even simultaneously. Yes you can actually have more than one "View" window open at the same time.

Now that we have change the identifier to "search", let us actually add some functionality of search to its respective view. We will begin by adding a new XIB file, this can be accomplished by right-clicking on the resources

folder and choosing add->New file. Under the User Interface category pick up a "View XIB", let us call it SearchIt.xib.

Next add a "UIViewController" subclass by right-clicking the classes folder in the "groups and file" pane and choosing it from the Cocoa Touch Classes category. We will call this one as SearchItController.m. Following this open up your just created xib file in the interface builder and add a "Search Bar View" to it. Position it at the top of the window with its width covering the entire screen.



Right-click in this area to launch the new file dialog box

Now we will need to tell the xib file as to which class it belongs, therefore open the Identity Inspector of SearchIt.xib and choose SearchItController as its Class. Next we will need to link it, therefore control-drag the "File's Owner" item from the "SearchIt.xib" window to the "view" item and select "view" itself from the pop-ip. Next control-drag the search bar view to the File's Owner item and select Delegate.

Now move back to your MainWindow.xib and select the "Tab Bar" item and open its Attributes Inspector widnow. Set the NIB name as SearchIt. Now you are done with the Interface builder part therefore save the files and quit it.

Back in your Xcode modify your SearchItController.h file as following:

```
#import <UIKit/UIKit.h>
@interface SearchItController : UIViewController
<UISearchBarDelegate> {
}
@end
```

```
  Following which open your SearchItController.m file and
modify it as follows:
  #import "SearchItController.h"
  @implementation SearchItController
  -   (void)searchBarSearchButtonClicked:(UISearchBar   *)
searchBar
    {        // To ensure your keyboard doesn't pop up
unexpectedly
        [searchBar resignFirstResponder];
    }
```

And we are done. Press [command] + [R] to build, run and launch the simulator. You will be greeted by a tab-bar screen with the search tab on the right most lower corner. Enjoy your freshly-baked application.

As an ending note we would like to remind you that incase you want your application to support rotation you just need to ensure that the following peice of code is not commented in your ViewController.m files and the return type is set to YES.

```
  - (BOOL)shouldAutorotateToInterfaceOrientation:(UIInter
faceOrientation)
      interfaceOrientation {
      return YES;
  }
```

7.4 Navigation-based application

These types of apps are typically designed for the settings menu or the playlist as in the iPod interface. Basically this is the substitute for "Menu" in a typical desktop application. Best suited in scenarios where you want to present data in a hierarchical manner.

Without any further delay fire up your Xcode and create a new application. This time, select a "Navigation-based" application. Let us call it Navi.

Now apart from the regular files that are there for any view-based application, you will notice that there are two new XIB files, one called the RootViewController.xib and the other titled as MainWindow.Xib

We will take the same approach as the one we took for Tab-bar application, we will first analyze, observe and understand the default template and then move on to modifying it. First open the NaviAppDelegate.h file, the contents would look something like the following.

```
  @interface     NaviAppDelegate     :     NSObject
```

```
<UIApplicationDelegate> {
      UIWindow *window;
  //Note that this time the subclass is called the //
UINavigationController, unlike any other code we have seen
before
      UINavigationController *navigationController;
  }
  @property (nonatomic, retain) IBOutlet UIWindow *window;
  @property     (nonatomic,     retain)     IBOutlet
UINavigationController
       *navigationController;
```

As explained in the comment that UINavigationController is a specialized View Controller provided by the SDK for the Menu driver applications.

Now open your NaviAppDelegate.m file

```
#import "NaviAppDelegate.h"
#import "RootViewController.h"
@implementation NaviAppDelegate
@synthesize window;
@synthesize navigationController;
 #pragma mark -
#pragma mark Application lifecycle
//The method below makes the view of the Navigation
Controller the //default visible view, as soon as the
application is loaded into //the cpu
- (void)applicationDidFinishLaunching:(UIApplication *)
application {
      [window addSubview:[navigationController view]];
      [window makeKeyAndVisible];
}
```

Since the interface builder section is primarily a GUI area, we open the MainWindow.xib. Here, you'll observe that instead of the blank view window as in a "view-based" application, this view already contains a "Navigations Controller" item. The Navigation Controller item contains within itself a single Navigation Bar as well as a default view from the "RootViewController" class.

Following the visibility of the view from the RootViewController class we open the RootViewController.xib file in the Interface builder. If you look closely at the "RootViewController.xib" window you will notice that it comes

preloaded with a table view (The table view will be discussed in detail in the next chapter). The table view is not a necessity for the RootViewController.xib file, but it is present so often that the sdk loads it by default.

So say you want to display the name of your friends over here, open up Xcode and start editing the RootViewController.m file:

```
#import "RootViewController.h"
@implementation RootViewController
NSMutableArray *listOfFriends; //declares an array by the name of listOfFriends
- (void)viewDidLoad {
        listOfFriends = [[NSMutableArray alloc] init];
       // Now we will populate the array with actual names
        [listOfFriends addObject:@"Friend 1"];
        [listOfFriends addObject:@" Radhika"];
        [listOfFriends addObject:@"John" ];
        [listOfFriends addObject:@"Adi"];
        [listOfFriends addObject:@"Rico"];
        [listOfFriends addObject:@"Marsh"];
        [listOfFriends addObject:@"Some Friend"];
        [listOfFriends addObject:@"Some more friends"];
        [listOfFriends addObject:@"Alien 1"];
        [listOfFriends addObject:@"Alien from Pluto"];
        // The Following variable sets the title of the navigation bar
        self.navigationItem.title = @"Friends";
      [super viewDidLoad];
}
- (void)dealloc {
     [listOfFriends release];
     [super dealloc];
}
```

Now we need to inform the UI how many items were there in the list and what should be the appearance of each cell this can be done by modifying two methods of the table view. Don't worry if you don't understand some of the code below, as you read the later chapters this will become clear to you.

Tableview:numberOfRowsInSection should resemble like this

```
- (NSInteger)tableView:(UITableView *)tableView
              numberOfRowsInSection:(NSInteger)section {
```

```
        return [listOfFriends count];
}
```

Tableview:cellForRowAtIndexPath would look like this

```
- (UITableViewCell *)tableView:(UITableView *)tableView
                    cellForRowAtIndexPath:(NSIndexPath
*)indexPath {   static NSString *CellIdentifier = @"Cell";
      UITableViewCell *cell = [tableView
           dequeueReusableCellWithIdentifier:CellIdentifi
er];
      if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UI
TableViewCellStyleDefault reuseIdentifier:CellIdentifier]
autorelease];
      }
       NSString *cellValue = [listOfFriends objectAtIndex:
  indexPath.row];
        cell.textLabel.text = cellValue;
      return cell;
}
```

That should be enough to make a single view Navigation-Bar application with a table view embedded into it. But that is not what we are looking for, where are our other views.

So to accomplish the above we will follow similar steps as we did when we manually added a view controller. Right-click on your Classes group and go to `File > Add Files`, from there choose UIViewController subclass and don't forget to tick mark the "With XIB for user interface" option to save time. Let us name it phoneViewController.

Now open the phoneViewController.xib file in the interface builder and drag-drop a "label view" onto the "view" window.

In the XCode edit your phoneViewController.h file as following:

```
#import <UIKit/UIKit.h>
@interface phoneViewController : UIViewController {
       IBOutlet UILabel *label;
       NSString *textSelected;
}
@property (nonatomic, retain) UILabel *label;
@property (nonatomic, retain) NSString *textSelected;
-(id) initWithTextSelected:(NSString *) text;
```

```
@end
```

Now we need to make the connections. So first control-drag the "File's Owner" item to the Label View and in the pop-up select "Label". This takes care of connecting the outlet.

Next Control-drag the "File's Owner" item to the "View" item. This links up the View Controller to the actual View.

Now we need to edit the phoneViewController.m file to add the code. Open it up in Xcode and edit it as follows:

```
#import "phoneViewController.h"
@implementation phoneViewController
@synthesize label;
@synthesize textSelected;
- (id) initWithTextSelected:(NSString *) text {
    self.textSelected = text;
    [label setText:[self textSelected]];
    return self;
}
- (void)viewDidLoad {
    [label setText:[self textSelected]];
// Change the title of the Next Navigation Bar
    self.title = @"Phone Number of Friend";
    [super viewDidLoad];
}
- (void)dealloc {
    [label release];
    [textSelected release];
    [super dealloc];
}
```

Following this modify your RootViewController.h file to display the following:

```
#import "phoneViewController.h"
@interface RootViewController : UITableViewController {
    phoneViewController *phoneViewController;
}

@property (nonatomic, retain) phoneViewController
*phoneViewController;
@end
```

And the RootViewController.m file to resemble the following

```
#import "RootViewController.h"
@implementation RootViewController
NSMutableArray *listOfFriends;
@synthesize phoneViewController;
```

Now finally edit your tableView:didSelectRowAtIndexPath: method, this will be responsible for actually showing any detail

```
- (void)tableView:(UITableView *)tableView
                  didSelectRowAtIndexPath:(NSIndexPath  *)
indexPath {
       NSUInteger row = [indexPath row];
     NSString *rowValue = [listOfFriends objectAtIndex:row];
       NSString *message = [[NSString alloc]
  // %@ works like %s and replaces it in run time with the
actual name // of the friend, whose row has been selected
                         initWithFormat:@"The Phone number
of your friend \"%@\" is not stored yet", rowValue];
       if (self.detailsViewController == nil)
       {
  // You would notice the use of nib instead of xib
           phoneViewController *d = [[phoneViewController
alloc]
                                initWithNibName:@"phoneView"
                                          bundle:[NSBundle
mainBundle]];
           self.detailsViewController = d;
           [d release];
       }
                               [self.detailsViewController
initWithTextSelected:message];
       [self.navigationController pushViewController:self.
detailsViewController
            animated:YES];
  }
```

That is it. You have just created a navigation-based application with multiple views and a table embedded inside it. Go ahead, treat yourself, build, run and simulate the application and enjoy the smooth transitions. To

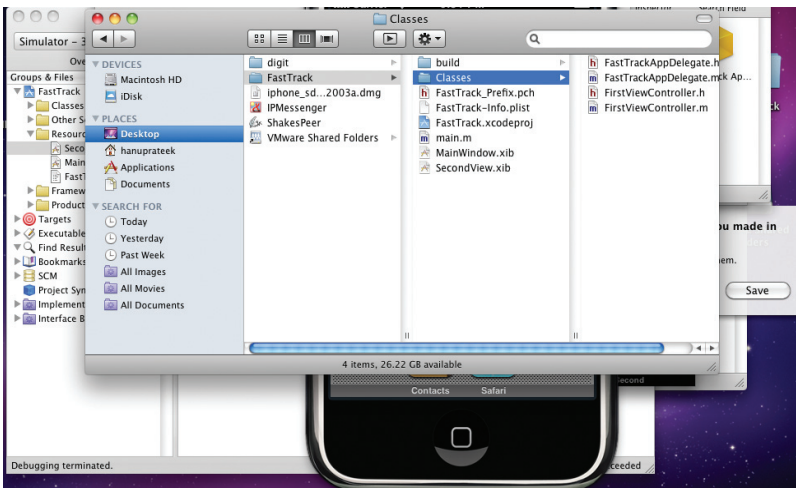
save time, we haven't actually entered the phone numbers so it will just show "The Phone number of your friend friend_name is not stored yet" when any friend is selected.

8 Handling files

We can go on and on about making straightforward applications that start, perform something and then end without saving any data for later use. But the fact is, most real world-applications need to store data in your application folder in flies, be it the files that you downloaded into memory from a remote server or user data meant for long-term storage that should persist even after the application shuts down and restarts.

8.1 About your application directories

Before we make an application that is capable of navigating into files and sub-directories, we need to understand the home directory concept of iPhone applications. Every time, you create an application, the following sub-directories are created and made available to you:



The typical libraries in an iPhone application

Applications: This is the directory that contains your application bundle, including the xib files generated by Xcode, localizations, executable code and the .app file that contains all other embedded resources.

Libraries: This folder is the parent to the Preferences directory of your application, where files written using the preferences APIs like

NSUserDefaults are stored.

Media: This is the folder where you will storing all the user's flat files and you can use it to store whatever media you want such as Documents and Images.

Root: It's highly unlikely that you will ever need to use this.

Tmp: This is the 'temporary' folder meant for short-lived, temporary files which are created while your application is running. This files in this folder generally have no long term relevance and it's a good programming practice to clear the contents of this folder everytime your application starts or shuts down to conserve space on the user's iPhone.

One catch with the iPhone file system management is the iPhone OS' sandbox model. Each application runs in a sandboxed environment, i.e., your application can only access its own folders. The rest of the file system is inaccessible. This means that your applications cannot work in synchronisation with another application and access its code and data. It cannot even access the user's music and videos, the system data, and everything else outside its own home folder. Any attempt to read from or write to these parts of the filesystem will fail.

8.2 Storing files in the documents folder

As we mentioned, the Documents folder is where you store all the files used by your application and user's flat files. The Library folder stores your preferences and application-specific settings. Lets see how to write a file in these folders:

Create a View-based application project in Xcode. We're calling it WorkOnFiles for future reference. Open the WorkOnFilesViewController.h file and add the following code to it:

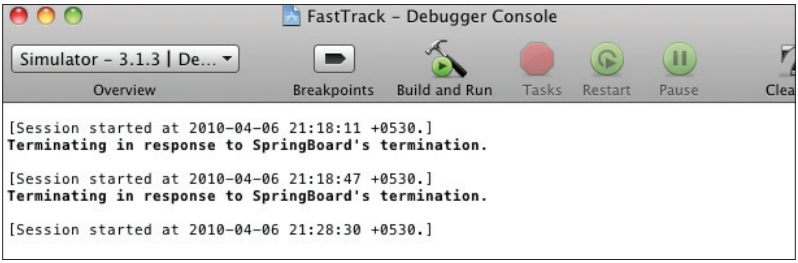
```
#import <UIKit/UIKit.h>
@interface WorkOnFilesViewController : UIViewController
{
}
-(NSString *) documentsPath;
-(NSString *) readFromFile:(NSString *) fi lePath;
-(void)      writeToFile:(NSString      *)      text
withFileName:(NSString *) fi lePath;
@end
```

Next, open the WorkOnFilesViewController.m file and add the following code:

```
#import "WorkOnFilesViewController.h"
@implementation WorkOnFilesViewController
//define the method that returns the path to the application's Documents directory
-(NSString *) documentsPath {
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
//This function creates a list of directory search paths and indicates that you are looking for //theDocuments folder. The NSUserDomainMask specifies to search in the application's //home directory.
NSString *documentsDir = [paths objectAtIndex:0];
//This command is used to obtain the path to the Documents folder, which is the first item //of the array you created. There is only one Documents folder for each application.
return documentsDir;
}
//read content from a specified file path
-(NSString *) readFromFile:(NSString *) fi lePath {
//check if the file exists
if ([[NSFileManager defaultManager] fi leExistsAtPath:fi lePath])
{
    //read the content of the file into an NSArray object.
NSArray *array = [[NSArray alloc] initWithContentsOfFile:filePath];
NSString *data = [[NSString alloc] initWithFormat:@"%@", [array objectAtIndex:0]];
//We extracted only the first element of the array because our file //just contains a single line of text. Account for this in an actual //implementation.
[array release];
return data;
}
else
return nil;
```

```
}
//This function creates an NSMutableArray and adds the text to be written into the specified file //path.
-(void) writeToFile:(NSString *) text withFileName:(NSString *) filePath {
NSMutableArray *array = [[NSMutableArray alloc] init];
[array addObject:text];
[array writeToFile:fi lePath atomically:YES];
//The automatically parameter here indicates that the file should first be written into a //temporary file, and then renamed to the file name specified. This guarantees that the file //is not corrupted in the case that your iPhone crashes while writing the file.
[array release];
}
// Implement viewDidLoad to do additional setup like creating the pathname for the file that you //want to save, writing a string into that file and then eventually reading it back and printing it in the //Debugger Console window after You've loaded the view from the xib.
- (void)viewDidLoad {
//formulate filename
NSString *fileName = [[self documentsPath]
stringByAppendingPathComponent:@"digit.txt"];
//write something to the file
[self writeToFile:@"Whatever string you want to write" withFileName:fileName];
//read it back
NSString *fileContent = [self readFromFile:fi leName];
//display the content read in the Debugger Console window
NSLog(fi leContent);
[super viewDidLoad];
}
```

Save the project and press `[command] + [R]` to start the application on the iPhone simulator. On the simulator, navigate to the Documents folder of your application from the Finder. You will see that a file called digit.txt is visible. This file will be present inside `/private/var/mobile/`



The Debugger Console Window

`Applications/<app_id>/Documents/digit.txt` on a real device. The contents of the file right now are:

```
< ?xml version="1.0" encoding="UTF-8"? >
< !DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd" >
< plist version="1.0" >
< array >
< string > Whatever string you want to write < /string >
< /array >
< /plist >
```

The Debugger Console window (accessible by `Shift + [command] + [R]`) shows that the application actually prints the string `Whatever string you want to write` because of the code in the viewDidLoad function.

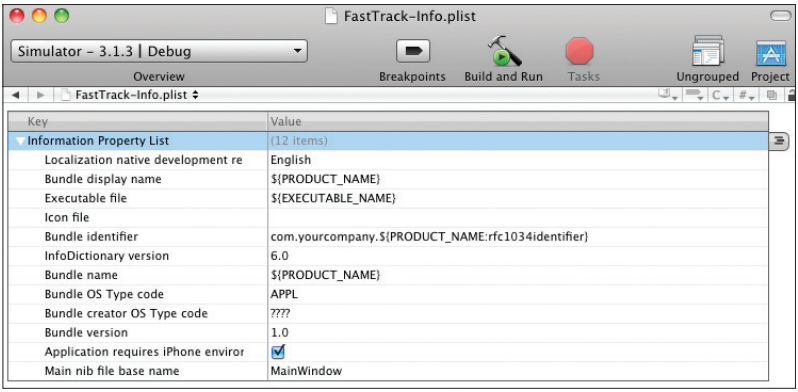
8.3 Using the temporary folder

You can store all the temporary files of your application in the tmp folder like in the previous instance, we created a temp file before writing it back it to the Documents folder. As a good programming practice, do not use this folder to store critical data because of iTunes does not back up this folder, and so it can lead to loss of data meant to be stored permanently.

To access the tmp folder, you first need to get its full path. Use the following function for this:

```
NSTemporaryDirectory() {
            -(NSString *) tempPath{
return NSTemporaryDirectory();
}
            }
```

This function will return the tmp folder `/private/var/mobile/`



The Property List Editor Interface in Xcode

`Applications/<app_id>/tmp/` on a real iPhone. To return the path of a particular file stored in the tmp folder, you can use the following command:

```
NSString *fileName = [[self tempPath]stringByAppendingPathComponent:@"digit.txt"];
```

8.4 Storing structured data

To store structured data (data categorised by key-value pairs) on an iPhone, you need to use Property Lists. These lists are stored as XML files and can be transferred across file systems, different iPhones and even across platforms or networks. Lets see how we can create and add a Property list on the Resources folder of your application using Xcode and then populate it with the built-in Property List Editor. We will also see how to programmatically retrieve values from the property list during runtime, make some changes to it and then write it back to another property list file.

8.5 Creating a property list

Right on the project that we created previously in this chapter and click on `Add > New File`. In the left pane, select Other and then choose the Property List template from the right pane. Click on Next and name it temp.list. Now, you can populate it from the editor that comes on the screen.

Next, we need to change the `viewDidLoad` method from the previous code as:

```
- (void)viewDidLoad {
//filename
```

```
  NSString *fileName = [[self documentsPath] stringByAppe
ndingPathComponent:@"digit.txt"];
  //write something to the file
  [self writeToFile:@"Whatever text you want to write"
withFileName:fileName];
  //read it back
  NSString *fileContent = [self readFromFile:fi leName];
  //display the content read in the Debugger Console
window
  NSLog(fileContent);
  //Try and locate the file named temp.plist in the
Documents folder and if found, get the path //to the
property list file
  NSString *plistFileName = [[self documentsPath]
  stringByAppendingPathComponent:@"temp.plist"];
  //if the property list file is found
  if ([[NSFileManager defaultManager] fileExistsAtPath:pl
istFileName])
  {
  //load the content of the property list file into an
NSDictionary object
  NSDictionary *dict = [[NSDictionary alloc]
  initWithContentsOfFile:plistFileName];
  //Enumerate through all the keys in the dictionary
object and print the titles for //each category in the
Debugger window
  for (NSString *category in dict)
  {
  NSLog(category);
  NSLog(@"--------------");
  NSArray *titles = [dict valueForKey:category];
  for (NSString *title in titles)
  {
  NSLog(title);
  }
  }
  [dict release];
  }
```

```
  else
   {
  //When the application is first run and the temp.plist
file is not available, we //need to load the property list
from the Resources folder
  NSString *pListPath = [[NSBundle mainBundle]
pathForResource:@"Temp"
  ofType:@"plist"];
  NSDictionary *dict = [[NSDictionary alloc]
  initWithContentsOfFile:pListPath];
  //Since we are making changes to the dictionary object,
we need to make a //mutable copy of the NSDictionary
object. This is necessary because once //the items are
populated from the property list, you cannot add any more
//content to the original NSDictionary object.
  NSMutableDictionary *copyOfDict = [dict mutableCopy];
  //Retrieve an array containing all the different
category keys
  NSArray *categoriesArray = [[copyOfDict allKeys]
  sortedArrayUsingSelector:@selector(compare:)];
  //Loop through all the keys to add more content in
the categories. We are //looping using the NSArray
object because it is not possible to add items to //the
NSMutableDictionary object while it is being enumerated.
  for (NSString *category in categoriesArray)
  {
  //get all the app titles in that category
  NSArray *titles = [dict valueForKey:category];
  //make a mutable copy of the array
  NSMutableArray *mutableTitles = [titles mutableCopy];
  //add a new title to the category
  [mutableTitles addObject:@"Temporary"];
  //set the array back to the dictionary object
  [copyOfDict setObject:mutableTitles forKey:category];
  [mutableTitles release];
  }
  //write the mutable dictionary object  to a file
  fileName = [[self documentsPath]
```

```
stringByAppendingPathComponent:@"temp.plist"];
[copyOfDict writeToFile:fi leName atomically:YES];
[dict release];
[copyOfDict release];
}
[super viewDidLoad];
}
```

Save the project and press [command] + [R] to start the application on the iPhone Simulator. As soon as the application starts, it creates a new .plist file in the Documents folder of your application. Viewing it in the Property List Editor will show that for each category, there's an item called Temporary. Running the application for the second time prints the contents of the .plist file to the Debugger Console Window (accessible by pressing Shift + [command] + [R]).

9 Handling Multi-Touch

The iPhone screen is a touch-sensitive display, which can detect multiple points of input and track them independently. It increases the iPhone's usability by allowing natural interaction between users and your applications. The applications can detect a wide range of gestures such as taps, swipes and pinches.

In this chapter, we will see how to detect touches in your application and then implement some of the other features that improve the interaction between the user and the application. We will also look at the underlying architecture that lets you detect gestures. The great part about UIKit is that it is very straightforward to add these types of interactions to our applications.

9.1 Detecting touches

Touch for the iPhone screen signifies a finger being placed on certain part of the screen. Multiple touches together make up a gesture, which is nothing but a sequence of events happening from the time you first touch the screen with one or more fingers and until none of them are in contact with the screen. An event containing information about the touch or touches that occurred is generated every time you interact with the multi-touch screen and your gestures are passed to the system inside it.

But, before we start detecting touches in our application, we need to acquaint ourselves with taps. A tap happens when the user touches the screen with a single finger and immediately lifts it up without moving it on the screen. Events handle the detection of touches in an iPhone and even keep track of the number of times a user has tapped, without any intervention from the developer as to the timing between multiple taps and differentiating between two single-taps and a double-tap. Lets dive into the architecture and learn how to detect a tap.

Create a View-based application project in Xcode. We are naming it TouchDetection. You can choose a different name, but make sure to account for it in the subsequent instructions. Add an image in the Resources folder by dragging and dropping. We will use it as the interaction point to detect a tap. The image we've used is Digit.jpeg. Next, open the TouchDetectionViewController.xib fine in the Interface Builder by double clicking on it. Choose the ImageView view in the View window and enlarge it to cover the entire View window. In the Attributes window, set the Image

property to Digit.jpeg. Open the TouchDetectionViewController.h file and add the following code :

```
#import <UIKit/UIKit.h>
@interface TouchDetectionViewController :
UIViewController {
    IBOutlet UIImageView *imageView;
    }
@property (nonatomic, retain) UIImageView *imageView;
@end
```

Now, go back to the Interface builder, [command] + click and drag the File's Owner item to the ImageView view and select it. Then open the TouchDetectionViewController.m file and add the following code to it:

```
#import "TouchDetectionViewController.h"
@implementation TouchDetectionViewController
@synthesize imageView;
//fired when the user's finger(s) touches the screen
-(void) touchesBegan: (NSSet *) touches withEvent:
(UIEvent *) event {
//get all touches on the screen
NSSet *allTouches = [event allTouches];
//compare the number of touches on the screen
switch ([allTouches count])
{
//single touch
case 1: {
//The UITouch object contains the tapcount property
which tells the //application whether the user has single
tapped or tapped more than once //on the screen
UITouch *touch = [[allTouches allObjects] objectAtIndex:0];
//compare the touches
switch ([touch tapCount])
{
//single tap
case 1: {
imageView.contentMode = UIViewContentModeScaleAspectFit;
} break;
//double tap
case 2: {
```

```
imageView.contentMode = UIViewContentModeCenter;
} break;
}
} break;
}
}
- (void)dealloc {
[imageView release];
[super dealloc];
}
```

Save the project and press [command] + [R] to start the application on the iPhone Simulator. Now, you can single click on the digit image to enlarge it and double-tap it to return to its original size.

9.2 Detecting multiple touches

Once you've the previous code up and running, implementing this feature won't be that big a problem. The concept is very similar. Open the same project and edit the touchesBegan:withEvent: method by inserting the following code in it:

```
-(void) touchesBegan: (NSSet *) touches withEvent:
(UIEvent *) event {
  //get all touches on the screen
  NSSet *allTouches = [event allTouches];
  //compare the number of touches on the screen
  switch ([allTouches count])
  {
  //single touch
  case 1: {
  //get info of the touch
  UITouch *touch = [[allTouches allObjects] objectAtIndex:0];
  //compare the touches
  switch ([touch tapCount])
  {
  //single tap
  case 1: {
  imageView.contentMode = UIViewContentModeScaleAspectFit;
  } break;
  case 2: {
```

```
  imageView.contentMode = UIViewContentModeCenter;
  } break;
  }
  } break;
  //double-touch
  case 2: {
  //get info of first touch
  UITouch    *touch1    =    [[allTouches    allObjects]
objectAtIndex:0];
  //get info of second touch
  UITouch    *touch2    =    [[allTouches    allObjects]
objectAtIndex:1];
  //get the points touched
  CGPoint touch1PT = [touch1 locationInView:[self view]];
  CGPoint touch2PT = [touch2 locationInView:[self view]];
  NSLog(@"Touch1: %.0f, %.0f", touch1PT.x, touch1PT.y);
  NSLog(@"Touch2: %.0f, %.0f", touch2PT.x, touch2PT.y);
  } break;
  }
  }
```

Save the project and press `[command] + [R]` to start the application on the iPhone Simulator. In the Simulator, if you press the Option key, two small circles will appear. Clicking on the screen now will simulate two fingers actually touching the screen of the device. You can also click the mouse and move it across to simulate pinching.

9.3 Using the pinch gesture

Pinch gesture involves using two fingers touching the screen and pinching by moving them closer to each other. In general, this translates to a zoom out action in the standard iPhone applications. With use knowing the technique of multi-touches, it should be fairly easy to implement. Use the same project that you created earlier. Open the TouchDetectionViewController.h file and edit it like:

```
  #import <UIKit/UIKit.h>
  @interface     TouchDetectionViewController     :
UIViewController {
  IBOutlet UIImageView *imageView;
  }
```

```
@property (nonatomic, retain) UIImageView *imageView;
-(CGFloat) distanceBetweenTwoPoints: (CGPoint)fromPoint
toPoint: (CGPoint)toPoint;
@end
```

Next open the TouchDetectionViewController.m file and edit it like:

```
#import "TouchDetectionViewController.h"
@implementation TouchDetectionViewController
@synthesize imageView;
CGFloat originalDistance;
-(CGFloat) distanceBetweenTwoPoints:(CGPoint)fromPoint
toPoint:(CGPoint)toPoint {
fl oat lengthX = fromPoint.x - toPoint.x;
fl oat lengthY = fromPoint.y - toPoint.y;
return sqrt((lengthX * lengthX) + (lengthY * lengthY));
}
//fired when the user's finger(s) touches the screen
-(void) touchesBegan: (NSSet *) touches withEvent:
(UIEvent *) event {
//get all touches on the screen
NSSet *allTouches = [event allTouches];
//compare the number of touches on the screen
switch ([allTouches count])
{
//single touch
case 1: {
//get info of the touch
UITouch *touch = [[allTouches allObjects] objectAtIndex:0];
//compare the touches
switch ([touch tapCount])
{
//single tap
case 1: {
imageView.contentMode = UIViewContentModeScaleAspectFit;
} break;
case 2: {
imageView.contentMode = UIViewContentModeCenter;
} break;
}
```

```
  } break;
  //double-touch
  case 2: {
  //get info of first touch
  UITouch    *touch1    =    [[allTouches    allObjects]
objectAtIndex:0];
  //get info of second touch
  UITouch    *touch2    =    [[allTouches    allObjects]
objectAtIndex:1];
  //get the points touched
  CGPoint touch1PT = [touch1 locationInView:[self view]];
  CGPoint touch2PT = [touch2 locationInView:[self view]];
  NSLog(@"Touch1: %.0f, %.0f", touch1PT.x, touch1PT.y);
  NSLog(@"Touch2: %.0f, %.0f", touch2PT.x, touch2PT.y);
  //record the distance made by the two touches
  originalDistance = [self distanceBetweenTwoPoints:touc
h1PT
  toPoint: touch2PT];
  } break;
  }
  }
  //fired when the user moved his finger(s) on the screen
  -(void)  touchesMoved:  (NSSet  *)  touches  withEvent:
(UIEvent *) event {
  //get all touches on the screen
  NSSet *allTouches = [event allTouches];
  //compare the number of touches on the screen
  switch ([allTouches count])
  {
  //single touch
  case 1: {
  } break;
  //double-touch
  case 2: {
  //get info of fi rst touch
  UITouch    *touch1    =    [[allTouches    allObjects]
objectAtIndex:0];
  //get info of second touch
```

```
  UITouch *touch2 = [[allTouches allObjects]
objectAtIndex:1];
  //get the points touched
  CGPoint touch1PT = [touch1 locationInView:[self view]];
  CGPoint touch2PT = [touch2 locationInView:[self view]];
  NSLog(@"Touch1: %.0f, %.0f", touch1PT.x, touch1PT.y);
  NSLog(@"Touch2: %.0f, %.0f", touch2PT.x, touch2PT.y);
  CGFloat currentDistance = [self distanceBetweenTwoPoints:
touch1PT
  toPoint: touch2PT];
  //zoom in
  if (currentDistance > originalDistance)
  {
  imageView.frame = CGRectMake(imageView.frame.origin.x
- 2,
  imageView.frame.origin.y - 2,
  imageView.frame.size.width + 4,
  imageView.frame.size.height + 4);
  }
  else {
  //zoom out
  imageView.frame = CGRectMake(imageView.frame.origin.x
+ 2,
  imageView.frame.origin.y + 2,
  imageView.frame.size.width - 4,
  imageView.frame.size.height - 4);
  }
  originalDistance = currentDistance;
  } break;
  }
  }
  - (void)dealloc {
  [imageView release];
  [super dealloc];
  }
```

Save the project and press `[command] + [R]` to start the application on the iPhone Simulator. Now your pinch gesture should work.

9.4 Using the drag gesture

This refers to as tapping an item on the screen and then dragging it by just moving your finger. It is very easy to implement. Open the same project and make the size of the ImageView smaller so that you can drag it across the screen. Then open the TouchDetectionViewController.m file and edit the touchesMoved:withEvent: method in this way:

```
  -(void) touchesMoved: (NSSet *) touches withEvent:
(UIEvent *) event {
  //get all touches on the screen
  NSSet *allTouches = [event allTouches];
  //compare the number of touches on the screen
  switch ([allTouches count])
  {
  //single touch
  case 1: {
  //get info of the touch
  UITouch *touch = [[allTouches
  allObjects] objectAtIndex:0];
  //check to see if the image is being touched
  CGPoint touchPoint = [touch locationInView:[self view]];
  if (touchPoint.x > imageView.frame.origin.x &&
  touchPoint.x < imageView.frame.origin.x +
  imageView.frame.size.width &&
  touchPoint.y > imageView.frame.origin.y &&
  touchPoint.y <imageView.frame.origin.y +
  imageView.frame.size.height) {
  [imageView setCenter:touchPoint];
  }
  } break;
  //double-touch
  case 2: {
  //get info of fi rst touch
  UITouch    *touch1    =    [[allTouches    allObjects]
objectAtIndex:0];
  //get info of second touch
  UITouch    *touch2    =    [[allTouches    allObjects]
objectAtIndex:1];
  //get the points touched
```

```
  CGPoint touch1PT = [touch1 locationInView:[self view]];
  CGPoint touch2PT = [touch2 locationInView:[self view]];
  NSLog(@"Touch1: %.0f, %.0f", touch1PT.x, touch1PT.y);
  NSLog(@"Touch2: %.0f, %.0f", touch2PT.x, touch2PT.y);
  CGFloat currentDistance = [self distanceBetweenTwoPoints:
touch1PT
  toPoint: touch2PT];
  //zoom in
  if (currentDistance > originalDistance)
  {
  imageView.frame  =  CGRectMake(imageView.frame.origin.x
- 2,
  imageView.frame.origin.y - 2,
  imageView.frame.size.width + 4,
  imageView.frame.size.height + 4);
  }
  else {
  //zoom out
  imageView.frame  =  CGRectMake(imageView.frame.origin.x
+ 2,
  imageView.frame.origin.y + 2,
  imageView.frame.size.width - 4,
  imageView.frame.size.height - 4);
  }
  originalDistance = currentDistance;
  } break;
  }
  }
```

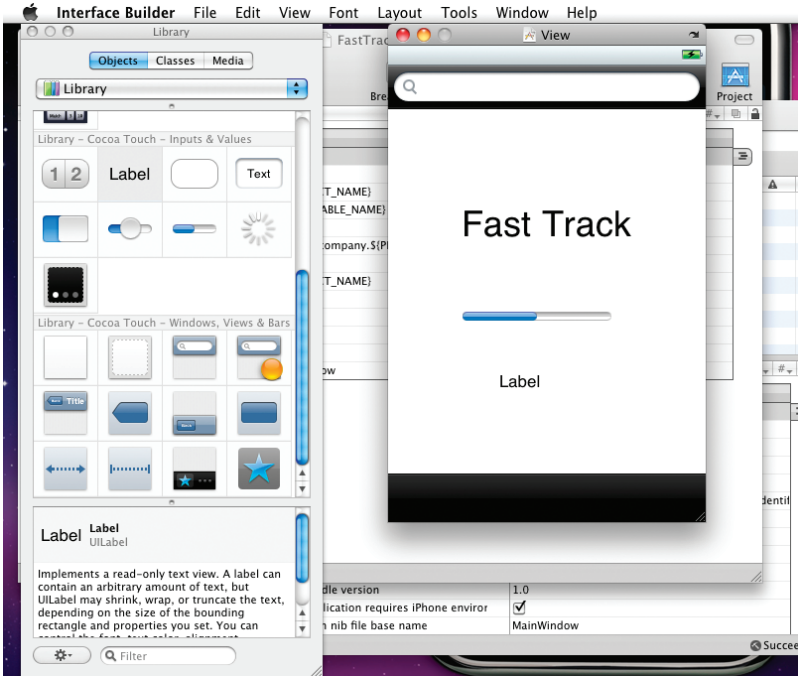
Save the project and press `[command] + [R]` to start the application on the iPhone Simulator. Now you can tap on the ImageView and drag it across the screen by simply moving the finger without lifting it off. In these similar ways, you can apply more gestures without any difficulty.

10 Animations

iPhone applications use the standard views made available by the iPhone SDK to interact with you. You can create animated interfaces which are visually appealing using the same views by binding them together with the help of a timer object to control transformations on views. These animations will not be as cool as those made using OpenGL and will not serve as gaming graphics, but they will definitely go on to make your application fun and visual.

10.1 The NSTimer class

The NSTimer class is used to create the timer object, which can then call a method that you specify at regular time intervals. This will effectively create a impression of animation if you keep updating an image. We will take the



The View after a Label, an image and a slider has been added

logo of Digit and display it on the screen at different places in a short interval making it look as if the logo is animated and moving around the screen. The frequency and the way it reacts on going to any one side will also be controlled by code.

Create a new View-based Application project in Xcode. We will call it AnimatedLogo. Now drag and drop the logo of Digit, i.e., digit.jpeg in the Resources folder in Xcode. If the Add dialog box opens up, just select Copy Item into Destination Group's Folder check box to make sure that a copy of image is copied into the project. Next, open the AnimatedLogoViewController.xib in the Interface Builder by double-clicking on it. Go to the View window, and drag an ImageView onto it. Also set its Image property to digit.jpeg. To control the animation, let's also add a Slider view and Label it with a Label view from the library. Make sure you set the Initial property of the Slider View to 0.01.

Open the AnimatedLogoViewController.h file now and declare the following fields, outlets and actions:

```
#import <UIKit/UIKit.h>
@interface AnimatedLogoViewController : UIViewController
{
IBOutlet UIImageView *imageView;
IBOutlet UISlider *slider;
CGPoint position;
NSTimer *timer;
float logoRadius;
}
@property (nonatomic, retain) UIImageView *imageView;
@property (nonatomic, retain) UISlider *slider;
-(IBAction) sliderMoved:(id) sender;
@end
```

Go back to the Interface Builder and connect the imageView, slider and view outlets to the sliderMoved action. Then open the AnimatedLogoViewController.m file and add the following code to it:

```
#import "AnimatedLogoViewController.h"
@implementation AnimatedLogoViewController
@synthesize imageView;
@synthesize slider;
//In the onTimer method, we will change the position
of the ImageView by setting its center property to a new
```

```
value.
  -(void) onTimer {
  imageView.center = CGPointMake(
  imageView.center.x + position.x,
  imageView.center.y + position.y);
  if (imageView.center.x > 320 - logoRadius || imageView.
center.x < logoRadius)
  position.x = -position.x;
  if (imageView.center.y > 460 - logoRadius || imageView.
center.y < logoRadius)
  position.y = -position.y;
  }
  - (void)viewDidLoad {
                    //Get the logo's effective radius,
which is nothing but half the width of it.
  //This value will be used to detect if the logo has
touched the edges of the //iPhone's screen
  logoRadius = imageView.frame.size.width/2;
  [slider setShowValue:YES];
  //Give values to the position variable to specify how
much the image must move at each timer interval. We
will move the image 10 pixes horizontally and 5 pixels
vertically.
  position = CGPointMake(10.0,5.0);
  //Specify the timer interval. We are linking it to the
value of the slider.
  timer = [NSTimer scheduledTimerWithTimeInterval:slider.
value
  target:self
  selector:@selector(onTimer)
  userInfo:nil
  repeats:YES];
  [super viewDidLoad];
  }
  -(IBAction) sliderMoved:(id) sender {
  [timer invalidate];
  timer = [NSTimer scheduledTimerWithTimeInterval:slider.
value
```

```
target:self
selector:@selector(onTimer)
userInfo:nil
repeats:YES];
}
- (void)didReceiveMemoryWarning {
[super didReceiveMemoryWarning];
}
- (void)dealloc {
[timer invalidate];
[imageView release];
[slider release];
[super dealloc];
}
@end
```

Save the project and press `[command] + [R]` to start the application on the iPhone Simulator. Now you will see that the digit logo moves around the screen. To vary the speed of the animation, you can move the slider around. Moving it to the right slows down the animation. If you move it a lot towards the right, you will notice that the animation slows down considerably and the movement of the logo becomes abrupt. You can make it smoother by animating the visual changes caused by setting the centre property of the View within an animation block in the following way:

```
[UIView beginAnimations:@"my_own_animation"
context:nil];
imageView.center = CGPointMake(
imageView.center.x + position.x,
imageView.center.y + position.y );
[UIView commitAnimations];
```

10.2 View transformations in detail

Now that we have seen how to simulate simple animations using the NSTimer class by constantly changing the position of the ImageView, let's see some of the transformation techniques supported by the iPhone SDK to achieve the same effect without just repositioning the view. We will cover some basic affine 2D transformations such as Translation, Rotation and Scaling.

10.3 Translation

It refers to moving the origin of the view by a certain amount which will be specified using the X and Y axes. iPhone SDK supports this by default with the transform property of the view. In the previous NSTimer example, we were changing the position of the view using its center property. Now, we will use its transform property. Open the AnimatedLogoViewController.h file and add the following code:

```
  //inside the AnimatedLogoViewController.h file
  CGPoint position;
  CGPoint translation;
  //inside the viewDidLoad method
  position = CGPointMake(10.0,5.0);
  translation = CGPointMake(0.0,0.0);
  -(void) onTimer {
  imageView.transform = CGAffi neTransformMakeTranslation(
translation.x, translation.y);
  translation.x = translation.x + position.x;
  translation.y = translation.y + position.y;
  if (imageView.center.x + translation.x > 320 - logoRadius
|| imageView.center.x + translation.x < logoRadius)
  position.x = -position.x;
  if (imageView.center.y + translation.y > 460 - logoRadius
|| imageView.center.y + translation.y < logoRadius)
  position.y = -position.y;
  }
```

10.4 Rotation

This is used to move the view by a specified angle. Let's modify the previous code only and make the logo rotate as it returns back after reaching an edge. Open the AnimatedLogoViewController.h file and add the declaration "float angle;" below the float logoRadius declaration. Also, change the AnimatedLogoViewController.m filke and edit it in the following way:

```
  -(void) onTimer {
  imageView.transform = CGAffi
neTransformMakeRotation(angle);
  angle += 0.02;
  if (angle>6.2857) angle = 0;
  imageView.center = CGPointMake(
```

```
  imageView.center.x + position.x,
  imageView.center.y + position.y);
  if (imageView.center.x > 320 - logoRadius || imageView.
center.x < logoRadius)
  position.x = -position.x;
  if (imageView.center.y > 460 - logoRadius || imageView.
center.y < logoRadius)
  position.y = -position.y;
  }
  - (void)viewDidLoad {
  angle = 0;
  logoRadius = imageView.frame.size.width/2;
  [slider setShowValue:YES];
  position = CGPointMake(12.0,4.0);
  timer = [NSTimer scheduledTimerWithTimeInterval:slider.
value
  target:self
  selector:@selector(onTimer)
  userInfo:nil
  repeats:YES];
  [super viewDidLoad];
  }
```

Save the project and press `[command] + [R]` to start the application on the iPhone Simulator. Now you will see that the digit logo rotates as it comes back after reaching any edge of the iPhone screen.

10.5 Scaling

It refers to changing the scale of the view by specifying the x and y factors. Set the transform property of the view to a CGAffineTransform data structure returned by the CGAffineTransformMakeScale() method like:

```
imageView.transform = CGAffineTransformMakeScale(angle
,angle);
```

This statement will make the digit logo in the previous example get bigger as it goes away from the edge before eventually getting back to its original size and continuing this pattern.

10.6 Animating with multiple images

Till now, we were using the ImageView view to display static images. Now

we will see how we can use it to display a series of images and alternate between them to create the illusion of a dynamic animation.

Created another View-based Application project in Xcode. We will call it Animation. Next, add a few images into the Resources folder by dragging and dropping them in the right place. If the Add dialog box opens up, just select Copy Item into Destination Group's Folder check box to make sure that a copy of image is copied into the project. Open the AnimationViewController.m file and add the following code in it :

```
- (void)viewDidLoad {
//Initialize the NSArray object with a few images.
NSArray *images = [NSArray
arrayWithObjects:
[UIImage imageNamed:@"digit.jpeg"],
[UIImage imageNamed:@"fasttrack.jpeg"],
[UIImage imageNamed:@9dot9.jpg"],
[UIImage imageNamed:@"digit_alt.jpeg"],
[UIImage imageNamed:@"dvd.jpeg"],
nil];
            //Instantiate a UIImageView object.
CGRect frame = CGRectMake(0,0,320,460);
UIImageView *imageView = [[UIImageView alloc]
initWithFrame:frame];
//Set the display mode of the ImageView and the
animationImages property
imageView.animationImages = images;
imageView.contentMode = UIViewContentModeScaleAspectFit;
//number of seconds taken by ImageView to display one
complete set of images.
imageView.animationDuration = 3;
//Specify how many times you want the animation to
occur.0 means indefinitely.
imageView.animationRepeatCount = 0;
[imageView startAnimating];
//Add the ImageView to the view.
[self.view addSubview:imageView];
[imageView release];
[super viewDidLoad];
}
```

Save the project and press [command] + [R] to start the application on the iPhone Simulator. You will see that the specified images are displayed in the ImageView view one by one on the iPhone screen.

11 Inbuilt applications

One of the reasons of iPhone's tremendous success is the number of built-in applications that come with it. These applications which include Contacts, Mail, SMS, Safari, Phone, and Calendar perform most of the tasks required by an average user. These applications can be programmatically invoked from within your application through the various API's provided in the iPhone SDK so that you can work synchronously with them and interact with them from within your application.

11.1 Accessing the photo library

The iPhone SDK allows you to programmatically invoke the Photos application and displays a UI which lets the user select pictures or videos from their albums and then you can use them in our application. This is made possible with the UIImagePickerController class of the SDK. To demonstrate this, let us create a new View-based Application in Xcode. We will name it PhotoLibraryAccess. Open the PhotoLibraryAccessViewController.xib in the Interface Builder by double-clicking on the file. Next, populate your View window with a Button View and an ImageView. Label the button as "Load a photo from the Library". Also, set the mode in the Attributes Inspector window of the ImageView to Aspect fit.

Now open the PhotoLibraryAccessViewController.h file and add the following source code in it:

```
##import <UIKit/UIKit.h>
@interface PhotoLibraryAccessViewController :
UIViewController
<UINavigationControllerDelegate,
UIImagePickerControllerDelegate> {
IBOutlet UIImageView *imageView;
UIImagePickerController *imagePicker;
}
@property (nonatomic, retain) UIImageView *imageView;
-(IBAction) btnClicked: (id) sender;
@end
```

Go back to the Interface Builder window and drag the File Owner's item to the ImageView vie by control clicking and dragging. Also select the ImageView. After this, control click on the Button view and drag it to the

File Owner's item select btnclicked:. Once you are done with all this, open the PhotoLibraryAccessViewControlller.m file and add the following source code in it:

```
#import "PhotoLibraryAccessViewController.h"
@implementation PhotoLibraryAccessViewController
@synthesize imageView;
- (void)viewDidLoad {
imagePicker = [[UIImagePickerController alloc] init];
[super viewDidLoad];
}
- (IBAction) btnClicked: (id) sender{
imagePicker.delegate = self;
imagePicker.sourceType =
UIImagePickerControllerSourceTypePhotoLibrary;
//show the Image Picker
[self presentModalViewController:imagePicker
animated:YES];
}
- (void)imagePickerController:(UIImagePickerController
*)picker didFinishPickingMediaWithInfo:(NSDictionary *)
info {
UIImage *image;
NSURL *mediaUrl;
mediaUrl = (NSURL *)[info valueForKey:UIImagePickerCont
rollerMediaURL];
if (mediaUrl == nil)
{
image = (UIImage *) [info valueForKey:UIImagePickerCont
rollerEditedImage];
if (image == nil)
{
//original image selected
image = (UIImage *)
[info valueForKey:UIImagePickerControllerOriginalIma
ge];
//display the image
imageView.image = image;
}
```

```
  else //edited image picked
  {
  //get the cropping rectangle applied to the image
  CGRect rect =
  [[info   valueForKey:UIImagePickerControllerCropRect]
CGRectValue ];
  //display the image
  imageView.image = image;
  }
  }
  else
  {
  //video picked
  }
  //hide the Image Picker
  [picker dismissModalViewControllerAnimated:YES];
  }
  - (void)imagePickerControllerDidCancel:(UIImagePickerCo
ntroller *)picker
  {
  //user did not select image/video; hide the Image Picker
  [picker dismissModalViewControllerAnimated:YES];
  }
  - (void)dealloc {
  [imageView release];
  [imagePicker release];
  [super dealloc];
  }
```

Save the project and press `[command] + [R]` to start the application on the iPhone Simulator. Once the application loads up, you will be presented with the button you named "Load a photo from the Library". Click on it and the iPhone Simulator will open up the photo album. If you select any picture here, it will get sent to your application by the event model and then displayed on the ImageView that you put on your application.

11.2 Accessing the camera

You can also access the camera of your iPhone (iPod Touch users will not get this feature) besides accessing the Photo Library programmatically. This

is also made possible using the same UIImagePickerController class of the iPhone SDK that we used in our last example. Since the implementation is fairly similar, we will just modify the previous project, instead of beginning from scratch. First of all, edit the PhotoLibraryAccessViewController.m file that you created by changing the source type of the Image picker to camera like:

```
-(IBAction) btnClicked: (id) sender{
imagePicker.delegate = self;
//invoke the camera
imagePicker.sourceType =
UIImagePickerControllerSourceTypeCamera;
imagePicker.allowsImageEditing = YES;
[self presentModalViewController:imagePicker
animated:YES];
}
```

Next open the PhotoLibraryAccessViewController.h file and declare two new methods that will help you with saving the image that you acquire from the camera into the application's Documents folder. We already covered how to do such tasks in the chapter on Handling Files.

The two new methods are:

```
- (NSString *) filePath: (NSString *) fileName;
- (void) saveImage;
```

These two methods now need to be defined in the PhotoLibraryAccessViewController.m file as:

```
- (NSString *) filePath: (NSString *) fileName {
NSArray *paths = NSSearchPathForDirectoriesInDomains(
NSDocumentDirectory, NSUserDomainMask, YES);
NSString *documentsDir = [paths objectAtIndex:0];
return [documentsDir stringByAppendingPathComponent:fi
leName];
}
- (void) saveImage{
//get the date from the ImageView
NSData *imageData =
[NSData dataWithData:UIImagePNGRepresentation(imageVi
ew.image)];
//write the date to file
```

```
[imageData writeToFile:[self fi lePath:@"MyPicture.
png"] atomically:YES];
}
```

You also need to add the MediaPlayer framework to your project. To accomplish this, just right-click on the Frameworks group in Xcode and click on `Add > Existing Frameworks` and select Frameworks/MediaPlayer.framework. Once you've done this, you can import this into your PhotoLibraryAccessViewController.h file with the `#import <MediaPlayer/MediaPlayer.h>` statement at the top of it.

After this, edit the PhotoLibraryAccessViewController.m file to look like this:

```
- (void)imagePickerController:(UIImagePickerController
*)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info {
UIImage *image;
NSURL *mediaUrl;
mediaUrl = (NSURL *)[info valueForKey:UIImagePickerCont
rollerMediaURL];
if (mediaUrl == nil)
{
image = (UIImage *) [info valueForKey:UIImagePickerCont
rollerEditedImage];
if (image == nil)
{
//original image selected
image = (UIImage *)
[info valueForKey:UIImagePickerControllerOriginalIma
ge];
//display the image
imageView.image = image;
//save the image captured
[self saveImage];
}
else
{
//edited image picked
//get the cropping rectangle applied to the image
CGRect rect =
```

```
  [[info   valueForKey:UIImagePickerControllerCropRect]
CGRectValue ];
  //display the image
  imageView.image = image;
  //save the image captured
  [self saveImage];
  }
  }
  else
  {
  //video picked
  MPMoviePlayerController        *player        =
[[MPMoviePlayerController alloc]
  initWithContentURL:mediaUrl];
  [player play];
  }
  //hide the Image Picker
  [picker dismissModalViewControllerAnimated:YES];
  }
```

Save the project and press `[command] + [R]` to start the application on the iPhone Simulator. Once the application loads up, you will be presented with the button you named "Load a photo from the Library". Clicking on it should open up the iPhone's camera application and you should be ready to take images and videos. If you capture any image here, it will get sent to your application by the event model and then displayed on the ImageView that you put on your application. If however, you take a video, it will be played back using the media player on your device.

11.3 Accessing the Mail application

You can send emails from within your application in two ways:

1. Build your own email client and make sure all the necessary protocols required to communicate with the email server and properly implemented.
2. Invoke the Mail application on the iPhone and use it to send email for you.

The first choice is obviously problematic, and cumbersome besides being a waste of developer resources. Lets see how we can go about with the second choice:

Create a new View-based Application project in Xcode. We will name it Email. Then open the EmailViewController.xib file in the Interface Builder

by double-clicking on it.

Drag and drop the following views in the View window:

- Labels (The to:, Subject: and main text: labels
- Text Fields (to enter the email address, subject and the actual email body)
- Button (Label it SendEmail)

Now open the EmailViewController.h file and enter the following source code in it:

```
#import <UIKit/UIKit.h>
@interface EmailViewController : UIViewController {
IBOutlet UITextField *to;
IBOutlet UITextField *subject;
IBOutlet UITextField *body;
}
@property (nonatomic, retain) UITextField *to;
@property (nonatomic, retain) UITextField *subject;
@property (nonatomic, retain) UITextField *body;
-(IBAction) btnSend: (id) sender;
@end
```

Now, go back to the Interface builder and do the following: Control-click and drag the File Owner's item to all the TextFiled views and then select the to, subject and email body labels. Control-click and drag the Button view on to the File Owner's item and select btnSendEmail.

You need to open the EmailViewController.m file after this and insert the following source code:

```
#import "EmailViewController.h"
@implementation EmailViewController
@synthesize to, subject, body;
- (void) sendEmailTo:(NSString *) toStr withSubject:
(NSString *) subjectStr withBody: (NSString *) bodyStr {
NSString *emailString = [[NSString alloc] initWithForma
t:@"mailto:?to=%@&subject=%@&body=%@",
[toStr stringByAddingPercentEscapesUsingEncoding:NSASCI
IStringEncoding],
[subjectStr stringByAddingPercentEscapesUsingEncoding:N
SASCIIStringEncoding],
[bodyStr stringByAddingPercentEscapesUsingEncoding:NSAS
CIIStringEncoding]];
[[UIApplication sharedApplication] openURL:[NSURL
```

```
URLWithString:emailString]];
  [emailString release];
  }
  -(IBAction) btnSend: (id) sender{
  [self sendEmailTo:to.text withSubject:subject.text
withBody:body.text];
  }
  - (void)dealloc {
  [to release];
  [subject release];
  [body release];
  [super dealloc];
  }
```

Save the project and press `[command] + [R]` to start the application on the iPhone Simulator. Now you will be presented with the interface you just created. Once you enter in all the relevant fields and click on SendEmail, the Mail application will be invoked with all the fields filled just as you did in your application by default. All you need to do now, is click on the Send button in Mail and you are done.

Accessing Safari

To invoke the default web browser of the iPhone, i.e., Safari from your application, you need to use the following code with the URL as per your preference. You can even take the URL from a text field entered by an user:

```
[[UIApplication sharedApplication] openURL:[NSURL
URLWithString: @"http://www.thinkdigit.com"]];
```

Accessing the Dialer

To make a phone call to a number specified by your application, you can invoke the iPhone's dialer with the number prefilled from within your application using the following source code:

```
[[UIApplication sharedApplication] openURL:[NSURL URLWi
thString:@"tel:9881xx3xxx"]];
```

Accessing SMS

To send an SMS using the SMS application on the iPhone with a prefilled number, you can invoke it from your application using the following source code:

```
[[UIApplication sharedApplication] openURL:[NSURL
URLWithString: @"sms: 9881xx3xxx 5"]];
```

Accessing the Contacts Application

To store any contact information that you gather from within your application, you need not create your own database. You can straightaway use the default Contacts application on your iPhone by programmatically invoking it from within the application. Lets demonstrate this with an actual project:

Create another View-based Application project in Xcode. We will Name it Contacts. Add the AddressBookUI frame work by right clicking on Frameworks and select `Add > Existing Frameworks` and choosing the Frameworks/Addressbook.framework and AddressBookUI framework.

Next open the ContactsViewController.xib file in the Interface Builder and drag and drop a Button view on the View window. Open the ContactsViewController.h file and add the following source code into it:

the ContactsViewController.h fi le:

```
#import <UIKit/UIKit.h>
#import <AddressBook/AddressBook.h>
#import <AddressBookUI/AddressBookUI.h>
@interface ContactsViewController :
UIViewController
<ABPeoplePickerNavigationControllerDelegate> {
}
-(IBAction) btnClicked: (id) sender;
@end
```

Go back to the Interface Builder and control-click + drag the Button view to the File Owner's item and select btnClicked:. Then open the ContactsViewController.m file and add the following source code:

```
#import "ContactsViewController.h"
@implementation ContactsViewController
-(IBAction) btnClicked: (id) sender{
ABPeoplePickerNavigationController *picker =
[[ABPeoplePickerNavigationController alloc] init];
picker.peoplePickerDelegate = self;
//display the People Picker
[self presentModalViewController:picker animated:YES];
[picker release];
}
```

```
- (void)peoplePickerNavigationControllerDidCancel:
(ABPeoplePickerNavigationController *)peoplePicker {
//hide the People Picker
[self dismissModalViewControllerAnimated:YES];
}
- (BOOL)peoplePickerNavigationController:
(ABPeoplePickerNavigationController *)peoplePicker
shouldContinueAfterSelectingPerson:(ABRecordRef)person
{
//get the First Name
NSString *str = (NSString *)ABRecordCopyValue(person,
kABPersonFirstNameProperty);
str = [str stringByAppendingString:@"\n"];
//get the Last Name
str = [str stringByAppendingString:(NSString *)
ABRecordCopyValue(
person, kABPersonLastNameProperty)];
str = [str stringByAppendingString:@"\n"];
//get the Emails
ABMultiValueRef emailInfo = ABRecordCopyValue(person,
kABPersonEmailProperty);
//iterate through the emails
for (NSUInteger i=0; i< ABMultiValueGetCount(emailIn
fo); i++) {
str = [str stringByAppendingString:
(NSString *)ABMultiValueCopyValueAtIndex(emailInfo,
i)];
str = [str stringByAppendingString:@"\n"];
}
//display the details
UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:@"Selected Contact"
message:str delegate:self
cancelButtonTitle:@"OK"
otherButtonTitles:nil];
[alert show];
[alert release];
//hide the People Picker
```

```
  [self dismissModalViewControllerAnimated:YES];
  Accessing the Contacts Application | 371
  return NO;
  }
  -        (BOOL)peoplePickerNavigationController:
(ABPeoplePickerNavigationController *)peoplePicker
  shouldContinueAfterSelectingPerson:(ABRecordRef)person
property:(ABPropertyID)property
  identifi er:(ABMultiValueIdentifi er)identifi er {
  [self dismissModalViewControllerAnimated:YES];
  return NO;
  }
```

Save the project and press `[command] + R` to start the application on the iPhone Simulator. The View that you just created will be shown to you. If you click on the button, the Contacts application will be invoked and if you select a Contact, its contact details will be shown forward to you in an Alert View.

12 Accessing the hardware

The only part we've not dealt in this book till now is how to access the iPhone's hardware. We've used the applications, camera, contact list etc already in our projects. Now in this chapter, we will show you how to use specialized classes in the iPhone SDK to access hardware devices, such as the accelerometer, GPS and how to obtain location information using GPS, cell towers and wireless hotspots.

Let's start with an application that uses the accelerometer for some purpose other than changing the rotation of the screen, which we've already shown you how to. The accelerometer of the iPhone allows the device to detect the orientation of the device and adapts the content to suit the new orientation. Even the camera relies on the accelerometer to tell it whether you are taking a picture in portrait or landscape mode. It measures the acceleration of the device relative to freefall in three different axes: X, Y,and Z. To make things clear, here's a table to show the values of X, Y and Z for various positions of the iPhone screen.

Position	X	Y	Z
Vertical upright position	0.0	-1.0	0.0
Landscape Left	1.0	0.0	0.0
Landscape Right	-1.0	0.0	0.0
Upside Down	0.0	1.0	0.0
Flat Up	0.0	0.0	-1.0
Flat Down	0.0	0.0	1.0

In general, the value of X will increase if the device is held upright and moved to the right quickly. Similarly, if moved to the left quickly, X will decrease. The same principle applies for the Y and Z directions too. The accelerometer used on the iPhone gives a maximum reading of about +/- 2.3G with a resolution of about 0.018 g.

Now that your concepts about the accelerometer are clear, lets get to work with some actual code. Make note that this code cannot be properly tested on the iPhone Simulator, so its recommended that you get a developer license and test it on an actual hardware. We will first begin with programmatically accessing the data returned by the accelerometer, which can then be used for motion detection algorithms in applications like games.

Create a new View-based Application project in Xcode. We will name it MotionDetection. Next open the MotionDetectionViewController.xib in the InterfaceBuilder by double-clicking on it and then populate the View window with 6 labels to test the X, Y and Z values.

Then Edit the MotionDetectionViewController.h file with the following source code:

```
#import <UIKit/UIKit.h>
@interface MotionDetectionViewController :
UIViewController
<UIAccelerometerDelegate> {
IBOutlet UILabel *labelX;
IBOutlet UILabel *labelY;
IBOutlet UILabel *labelZ;
}
@property (nonatomic, retain) UILabel *labelX;
@property (nonatomic, retain) UILabel *labelY;
@property (nonatomic, retain) UILabel *labelZ;
@end
```

After this, go back to the Interface builder and control click and drag the File Owner's item to each of the three label views that you decide to detect values and select labelX, labelY and labelZ. After this you are ready to write the actual code to test the values in the MotionDetectionViewController.m file:

```
#import "MotionDetectionViewController.h"
@implementation MotionDetectionViewController
@synthesize labelX, labelY, labelZ;
- (void)viewDidLoad {
UIAccelerometer     *acc     =     [UIAccelerometer
sharedAccelerometer];
acc.delegate = self;
acc.updateInterval = 1.0f/60.0f;
[super viewDidLoad];
}
- (void)accelerometer:(UIAccelerometer *) acc
didAccelerate:(UIAcceleration *)acceleration {
NSString *str = [[NSString alloc] initWithFormat:@"%g",
acceleration.x];
labelX.text = str;
```

```
  str = [[NSString alloc] initWithFormat:@"%g",
acceleration.y];
  labelY.text = str;
  str = [[NSString alloc] initWithFormat:@"%g",
acceleration.z];
  labelZ.text = str;
  [str release];
  }
  - (void)dealloc {
  [labelX release];
  [labelY release];
  [labelZ release];
  [super dealloc];
  }
```

Save the project and press `[command] + [R]` to start the application on the iPhone Simulator. The labels will now show real time orientation values of the X,Y and Z axes. So in concept, you know how to use them. Next is the part of writing algorithms in your application to use these. But we will leave that to your creativity.

We will however, explain how to detect shakes using the Shakes API. Again, create a new View-based Application project in Xcode. We will name it ShakeDetect. Open the ShakeDetectViewController.xib in the Interface Builder by double-clicking on it and populate the View window with a Text Field and a DatePicker view. Then open the ShakeDetectViewController.h file and write in the following source code:

```
  #import <UIKit/UIKit.h>
  @interface ShakeDetectViewController : UIViewController
{
  IBOutlet UITextField *textField;
  IBOutlet UIDatePicker *datePicker;
  }
  @property (nonatomic, retain) UITextField
  *textField;
  @property (nonatomic, retain) UIDatePicker
  *datePicker;
  -(IBAction) doneEditing: (id) sender;
  @end
```

After this, you need to go back to the Interface Builder and control-

click+drag the File Owner's item to the TextField view and select textField. Also, control-click+drag the FileOwner's item to the DatePicker view and select datePicker. Then, right-click on the TextField view and connect its Did End onExit event to the File Owner's item by Selecting doneEditing.

Now comes the actual coding part where you will edit the ShakeDetectViewController.m file with the following code:

```
#import "ShakeDetectViewController.h"
@implementation ShakeDetectViewController
@synthesize textField, datePicker;
- (void) viewDidAppear:(BOOL)animated
{
[self.view becomeFirstResponder];
[super viewDidAppear:animated];
}
- (IBAction) doneEditing: (id) sender {
//when keyboard is hidden, make the view the first
responder or else the //Shake API will not work
[self.view becomeFirstResponder];
}
-       (void)motionBegan:(UIEventSubtype)motion
withEvent:(UIEvent *)event {
if (event.subtype == UIEventSubtypeMotionShake )
{
NSLog(@"motionBegan:");
}
}
-      (void)motionCancelled:(UIEventSubtype)motion
withEvent:(UIEvent *)event {
if (event.subtype == UIEventSubtypeMotionShake )
{
NSLog(@"motionCancelled:");
}
}
-       (void)motionEnded:(UIEventSubtype)motion
withEvent:(UIEvent *)event {
if (event.subtype == UIEventSubtypeMotionShake )
{
NSLog(@"motionEnded:");
```

```
}
}
- (void)dealloc {
[textField release];
[datePicker release];
[super dealloc];
}
```

After this, add the UIView subclass template to the Classes group in Xcode by right clicking on the Classes group and going to `Add > New File`. We named it Shake.m and edited its source code to look like this:

```
#import "Shake.h"
@implementation Shake
- (id)initWithFrame:(CGRect)frame {
if (self = [super initWithFrame:frame]) {
// Initialization code
}
return self;
}
- (void)drawRect:(CGRect)rect {
// Drawing code
}
- (void)dealloc {
[super dealloc];
}
- (BOOL)canBecomeFirstResponder {
return YES;
}
@end
```

12.1 GPS

Using the GPS receiver on your iPhone, you can quickly find your location using the GPS satellites that have been deployed all around the Earth by USA. The iPhone also uses cell phone tower triangulation to get the co-ordinates faster than normal. This makes the location detection possible even while you are indoor where signals of GPS satellites are close to nil. The CoreLocation framework in the iPhone SDK makes all these services available to normal user-generated applications, like the ones we've been writing all throughout this book. Let us see how to go about it.

Create a new View-based Application project in Xcode. We will name it UsingGPS. Next open the UsingGPSViewController.xib in the InterfaceBuilder by double-clicking on it and then populate the View window with three labels and three text field views. Also, add the CoreLocation framework to your project by right clicking on the Frameworks group in Xcode and selecting Add Existing Framework Framework/CoreLocation.framework

Then Edit the UsingGPSViewController.h file with the following source code:

```
#import <UIKit/UIKit.h>
#import <CoreLocation/CoreLocation.h>
@interface UsingGPSViewController : UIViewController
<CLLocationManagerDelegate> {
IBOutlet UITextField *latitudeTextField;
IBOutlet UITextField *longitudeTextField;
IBOutlet UITextField *accuracyTextField;
CLLocationManager *lm;
}
@property (retain, nonatomic) UITextField
*latitudeTextField;
@property (retain, nonatomic) UITextField
*longitudeTextField;
@property (retain, nonatomic) UITextField
*accuracyTextField;
@end
```

After this, go back to the Interface builder and control click and drag the File Owner's item to each of the three TextField views and select latitudeTextField, longitudeTextField and accuracyTextField. After this you are ready to write the actual code to test the values in the UsingGPSViewController.m file:

```
#import "UsingGPSViewController.h"
@implementation UsingGPSViewController
@synthesize latitudeTextField, longitudeTextField,
accuracyTextField;
- (void) viewDidLoad {
lm = [[CLLocationManager alloc] init];
if ([lm locationServicesEnabled]) {
```

```
  lm.delegate = self;
  lm.desiredAccuracy = kCLLocationAccuracyBest;
  lm.distanceFilter = 1000.0f;
  [lm startUpdatingLocation];
  }
  }
  - (void) locationManager: (CLLocationManager *) manager
  didUpdateToLocation: (CLLocation *) newLocation
fromLocation: (CLLocation *) oldLocation{
  NSString *lat = [[NSString alloc] initWithFormat:@"%g",
  newLocation.coordinate.latitude];
  latitudeTextField.text = lat;
  NSString *lng = [[NSString alloc] initWithFormat:@"%g",
  newLocation.coordinate.longitude];
  longitudeTextField.text = lng;
  NSString *acc = [[NSString alloc] initWithFormat:@"%g",
  newLocation.horizontalAccuracy];
  accuracyTextField.text = acc;
  [acc release];
  [lat release];
  [lng release];
  }
  - (void) locationManager: (CLLocationManager *) manager
didFailWithError: (NSError *) error {
  NSString *msg = [[NSString alloc] initWithString:@"Error
obtaining location"];
  UIAlertView *alert = [[UIAlertView alloc]
  initWithTitle:@"Error"
  message:msg
  delegate:nil
  cancelButtonTitle: @"Done"
  otherButtonTitles:nil];
  [alert show];
  [msg release];
  [alert release];
  }
  - (void) dealloc{
  [lm release];
```

```
[latitudeTextField release];
[longitudeTextField release];
[accuracyTextField release];
[super dealloc];
}
```

Save the project and press [command] + R to start the application on the iPhone Simulator. This will show you the location of the place that the Simulator has locked the GPS on to. To test it in a real world scenario, you need to try this on a real iPhone.

Now that you can obtain the values of the position that you are at, it would be interesting to use this with the MapKit API to open the Google Map in your application with that location as the destination. Lets see how to go about it.

We will use the same project we created last. Just add a Button view to the View window in the UsingGPSViewController.xib file and add the MapKit framework in the same way that you added the CoreLocation framework. Then edit the UsingGPSViewController.h file to:

```
#import <UIKit/UIKit.h>
#import <CoreLocation/CoreLocation.h>
#import <MapKit/MapKit.h>
@interface UsingGPSViewController : UIViewController
<CLLocationManagerDelegate> {
IBOutlet UITextField *accuracyTextField;
IBOutlet UITextField *latitudeTextField;
IBOutlet UITextField *longitudeTextField;
CLLocationManager *lm;
MKMapView *mapView;
}
@property (retain, nonatomic) UITextField
*accuracyTextField;
@property (retain, nonatomic) UITextField
*latitudeTextField;
@property (retain, nonatomic) UITextField
*longitudeTextField;
-(IBAction) btnViewMap: (id) sender;
@end
```

Next, go back to the Interface builder and Control-click + drag the Button view to the File Owner's item and select btnViewMap. Also, edit the

UsingGPSViewController.m file to handle the event:

```
-(IBAction) btnViewMap: (id) sender {
[self.view addSubview:mapView];
}
- (void) viewDidLoad {
lm = [[CLLocationManager alloc] init];
lm.delegate = self;
lm.desiredAccuracy = kCLLocationAccuracyBest;
lm.distanceFilter = 1000.0f;
[lm startUpdatingLocation];
mapView = [[MKMapView alloc] initWithFrame:self.view.
bounds];
mapView.mapType = MKMapTypeHybrid;
}
- (void) locationManager: (CLLocationManager *)
manager didUpdateToLocation: (CLLocation *) newLocation
fromLocation: (CLLocation *) oldLocation{
NSString *lat = [[NSString alloc] initWithFormat:@"%g",
newLocation.coordinate.latitude];
latitudeTextField.text = lat;
NSString *lng = [[NSString alloc] initWithFormat:@"%g",
newLocation.coordinate.longitude];
longitudeTextField.text = lng;
NSString *acc = [[NSString alloc] initWithFormat:@"%g",
newLocation.horizontalAccuracy];
accuracyTextField.text = acc;
[acc release];
[lat release];
[lng release];
MKCoordinateSpan span;
span.latitudeDelta=.005;
span.longitudeDelta=.005;
MKCoordinateRegion region;
region.center = newLocation.coordinate;
region.span=span;
[mapView setRegion:region animated:TRUE];
}
- (void) dealloc{
```

```
[mapView release];
[lm release];
[latitudeTextField release];
[longitudeTextField release];
[accuracyTextField release];
[super dealloc];
}
```

Save the project and press `[command] + [R]` to start the application on the iPhone Simulator. When you click on the View Map button that you created, you will see that a map will be displayed with the location set to your present location, which in the case of the Simulator application is fixed. You will need an actual iPhone implementation to see if its actually working correctly.